

yesod

A FOUNDATION FOR BUILDING RELIABLE SOFTWARE

PRINCIPLES • PRACTICES • PATTERNS

Building and Operating an Autonomous Software Factory

From durable intent to gated, deployed code

Architecture · Operations · Governance · Roadmap

Source-grounded edition · November 22, 2026

Revision 1d5ab70

Stephen

Contents

About This Edition **1**
What Changed Since August . . . 2
What Is New, Rewritten and Revised . . . 2
Which Facts Win . . . 3
How Claims Are Marked . . . 3
How This Edition Was Made . . . 4
How to Read This Book . . . 4

Preface: A Factory, Not a Chat Room **6**

Prologue: The Accidental Factory **8**
Summer: Scar Tissue . . . 8
Autumn: The Factory Turns on Itself . . . 9
The Pattern . . . 10

I. One Change Through the Factory **12**

1. Follow the Change **13**
1.1. Captured, Then Planned . . . 14
1.2. Approved, Armed, Claimed . . . 14
1.3. One Child at a Time . . . 15
1.4. Held at the Door . . . 15
1.5. Attempt, Candidate, Gate . . . 16
1.6. Landing, and What Comes After . . . 16
1.7. One Door, Precisely Stated . . . 17
1.8. The Proof Chain . . . 18

2. Judgment and Mechanism **21**
2.1. The Control Plane . . . 21
2.1.1. Bootstrapping a role with `yesod prime` . . . 22
2.1.2. Authority: what code enforces and what is convention . . . 23
2.1.3. The planning and dispatch boundary . . . 24
2.2. The Execution Plane . . . 25
2.3. The Integration Plane . . . 25
2.4. The Data Plane . . . 26
2.5. Why Determinism Matters . . . 26

3. The Two Currencies **29**
3.1. Notes: Capturing Intent . . . 29
3.1.1. Status: how far the work has progressed . . . 30
3.1.2. Disposition: whether automation may act . . . 30
3.1.3. Arming: the third axis . . . 31
3.2. Beads: Planning the Work . . . 32
3.2.1. Where beads live . . . 32
3.2.2. Why every call is routed . . . 33
3.2.3. Plan-bound leaves . . . 33
3.3. Two Dependency Layers . . . 34
3.3.1. What clears a dependency . . . 35
3.3.2. Merged but not deployed parks the dependents . . . 35
3.4. Dispatch Lanes, Hosts and Affinity . . . 36
3.5. Four State Machines, Not One . . . 36

4. The Registry Beneath the Factory **39**
4.1. Tools as Durable Context . . . 39
4.2. Asking the Registry . . . 40
4.3. Processes as Molecules . . . 41
4.4. Composition Creates a Feedback Network . . . 42
4.5. Topics, and the Bulletin Over Them . . . 42
4.6. Onboarding a Repository . . . 43
4.7. The Worker Canon . . . 44
4.8. More Than a Factory Front End . . . 45

II. From Intent to Merge **47**

5. Capture, Plan, Deliver **48**
5.1. Three Fields, One Note . . . 48
5.2. The Path and Its Authorities . . . 49
5.3. One Note, Traced . . . 49
5.4. Planning Is a Durable Job . . . 50
5.5. The Plan Contract . . . 51
5.6. Bound to a Revision and a Fingerprint . . . 52
5.7. Approval Checks the Plan, Not the Approver . . . 53
5.8. Arming Is Automatic . . . 54
5.9. Claimable Is a Table . . . 54
5.10. Holds and the Admissions Pause . . . 55
5.11. What This Stage Guarantees . . . 56

6. The Runner **59**
6.1. Startup and the Wake . . . 59
6.2. One Poll Cycle . . . 59
6.3. The Claim . . . 60
6.4. The Worker Pool . . . 61
6.5. Worktrees and the Accumulating Branch . . . 62

6.6. Timers That Define Failure . . . 62
6.7. The Run Ledger and Three Heartbeats . . . 63
6.8. The Watchdog That Killed the Work . . . 64

7. Governed Agent Execution 68
7.1. The Worker Canon . . . 68
7.1.1. Plan-Bound Leaves . . . 69
7.2. The Launch Environment . . . 69
7.3. Limits Around the Process . . . 70
7.4. Proof of Work . . . 71
7.5. Ending Without a Commit . . . 72
7.6. The Blind Spot: Committed, Closed, Untested . . . 72
7.7. Execution Providers, and What Is Dormant . . . 73
7.8. Failure on Two Axes, and Bounded Retries . . . 74

8. The Merge Controller 78
8.1. The Vocabulary of a Merge . . . 78
8.2. The Good Case . . . 79
8.3. When the Gate Says No . . . 81
8.4. When the Owner Stalls . . . 82
8.4.1. The Second Fence . . . 82
8.5. When `main` Moved Underneath . . . 83
8.6. When Another Change Wants the Same Branch . . . 83
8.7. When the Merge Itself Conflicts . . . 84
8.8. Before the Gate . . . 85
8.9. When the Door Is Closed . . . 86
8.10. When the Process Simply Dies . . . 87
8.11. The Tick, Assembled . . . 87

9. The Deterministic Gate 91
9.1. The Vocabulary of a Gate . . . 91
9.2. Four Providers . . . 92
9.3. Why the Gate Came Home . . . 93
9.4. The Good Case: One Union . . . 93
9.5. The Demon and Its Boundary . . . 95
9.6. When the Tests Fail . . . 96
9.7. When the Gate Cannot Run . . . 97
9.8. Elasticity: What Is Live and What Is Staged . . . 98

III. From Merge to Running Code 103

10. Merged Is Not Deployed 104
10.1. The Gap . . . 104
10.2. Four Installs That Drift . . . 105
10.3. August: One Transaction Across Four Hosts . . . 105
10.4. September: Seats Deploy Themselves, the Controller Does Not . . . 106

10.5. Why Mutating a Long-Lived Host Kept Failing . . . 107
10.5.1. The deployment step wedged itself . . . 107
10.5.2. Code arrived before its schema . . . 108
10.5.3. Replaying old schema statements . . . 108
10.5.4. Something else migrated production . . . 109
10.6. A State for the Gap . . . 109
10.7. October: The Controller Tries to Promote Itself . . . 110
10.8. The Dependant Problem . . . 111
10.9. Schema Changes Become Opt-In and Missing-Only . . . 111
10.10. The Decision of 2026-10-13 . . . 112

11. Deploy Is a Host Swap 116
11.1. Why a Swap . . . 116
11.2. The Vocabulary of a Swap . . . 116
11.3. What Is Written Down . . . 117
11.4. What Starts a Swap . . . 118
11.5. One Swap, Step by Step . . . 119
11.5.1. Before any machine exists (steps 1 and 2) . . . 120
11.5.2. Birth, enrollment and qualification (steps 3 to 5) . . . 121
11.5.3. The handover (steps 6 to 9) . . . 121
11.5.4. The point of no return (step 10) . . . 122
11.5.5. After the boundary (steps 11 to 15) . . . 123
11.6. The Invariants . . . 124
11.7. Adopting a Hand-Born Controller . . . 125
11.8. The Dry Run . . . 125
11.9. When a Swap Fails . . . 126
11.9.1. Before the handover (steps 1 to 5) . . . 126
11.9.2. In the handover, before the provider handover . . . 126
11.9.3. After the provider handover, before address release . . . 126
11.9.4. After address release . . . 127
11.10. Proven Where . . . 127
11.10.1 The controllers so far . . . 128
11.10.2 The attempts . . . 128
11.10.3 The evening of 2026-11-19 . . . 128
11.10.4 Where it stands . . . 130

12. Born, Not Repaired 134
12.1. Why Rebirth . . . 134
12.2. The Seven Steps . . . 135
12.2.1. What “generation” means . . . 136
12.3. One Birth, Traced: a Gate VM . . . 137
12.4. Runner Birth . . . 139
12.5. Controller Birth . . . 139
12.6. Credentials Arrive Last . . . 140
12.7. Backups, Blobs and Retention . . . 142
12.8. The Topology Today . . . 142

IV. Operating the Factory 146

13. Seeing the Whole System 147
13.1. Telemetry Is Downstream of Truth . . . 147
13.2. The Event Spine . . . 147
13.3. The Exporter . . . 149
13.4. The Attempts Nobody Could See . . . 149
13.5. The Four Streams That Carry Diagnosis . . . 150
13.5.1. One Red Gate, Four Records . . . 151
13.6. One Health Projection . . . 151
13.7. Deployment Observation . . . 153
13.8. Eight Silent Failures in One Day . . . 153
13.9. The Backend . . . 154
13.10. Compliance as a Query . . . 154
13.11. The Evidence Map . . . 154
13.12. A Read-Only Diagnostic Rhythm . . . 155

14. Failure Is a State Machine 159
14.1. What Held and What Did Not . . . 159
14.2. The Vocabulary . . . 160
14.3. Class 1: Execution . . . 160
14.4. Class 2: Planning . . . 161
14.5. Class 3: Integration . . . 162
14.6. Class 4: Gate . . . 162
14.7. Class 5: Deployment and Replacement . . . 163
14.8. Class 6: Host and Infrastructure . . . 164
14.9. Class 7: Coordination . . . 164
14.10. Reading Across the Classes . . . 165

15. The Reliability Contracts 168
15.1. The Simple Example That Started It . . . 168
15.2. The Five Rules, Named . . . 169
15.3. Recovery Is a Contract Too . . . 169
15.4. History: The August Campaign . . . 170
15.4.1. One Row, Read Closely . . . 171
15.4.2. What the Campaign Showed, and What It Did Not . . . 172
15.5. The November Footnote to Durable Completion . . . 172
15.6. Five Rules the Autumn Paid For . . . 173
15.7. Where the Evidence Lives . . . 174
15.8. What a Reliability Contract Is . . . 175

16. Cost-Aware Dispatch 178
16.1. Three Things Called “Lane” . . . 178
16.2. How a Note Gets Its Dispatch Lane Today . . . 179
16.3. Cost Capture in Three Layers . . . 180
16.4. The Wrong Metrics . . . 181
16.5. Balance Reads, Floors and Parking . . . 181

16.6. The Spend Caps That Are Real . . . 182
16.7. Built, With No Production Caller . . . 183
16.8. What Would Have to Exist . . . 184

V. The Organization Around the Factory 187

17. The Agent Organization 188
17.1. Two Layers of Roles . . . 188
17.1.1. Why the second layer emerged . . . 190
17.2. Identity, Roster and Liveness . . . 190
17.3. Mail and What It Proves . . . 191
17.4. Why Coordination Moved to the Board . . . 192
17.4.1. How the board works . . . 192
17.4.2. Decision markers . . . 193
17.5. Role Directories and Durable Memory . . . 194
17.6. Subagents Never Run Tests . . . 194
17.7. The Isolated Build Seat as a Control . . . 195
17.8. A Rule Registry That States Its Own Assurance . . . 195
17.9. The Role Table . . . 196

18. Signed Human Authority 201
18.1. The Problem . . . 201
18.2. Where Human Authority Enters in Code . . . 202
18.3. Attention and Permission . . . 203
18.4. What a Grant Covers . . . 204
18.5. How a Request Is Prepared . . . 205
18.6. One Release Window, End to End . . . 205
18.7. The Rules and the Mistakes Behind Them . . . 207
18.7.1. A signer added mid-ask (2026-10-24) . . . 207
18.7.2. Parking in silence (2026-10-27) . . . 207
18.7.3. Signing a summary instead of the scope (2026-11-08) . . . 207
18.7.4. The mayor wrote the patch (2026-11-08) . . . 208
18.7.5. Approvals inside an open window (2026-11-09, FINDING #65) . . . 208
18.7.6. Four refusals in one day (2026-11-10) . . . 208
18.7.7. The missed five-minute cue (2026-11-10) . . . 209
18.7.8. Three taps lost to CLI details (2026-11-13) . . . 209
18.7.9. Green dry-run, red production (mid-November) . . . 210
18.7.10. Four reviewers and zero executions (2026-11-16 and 17) . . . 210
18.7.11. Nineteen of nineteen (2026-11-19, FINDINGS #128 and #130) . . . 210
18.8. The Cost . . . 210

19. What Failed, August to November 214
19.1. The Shape of the Autumn . . . 214
19.2. Late August: Contracts and Re-anchoring . . . 215
19.3. September: The Gate Comes Home . . . 216
19.4. October: Merged, Not Deployed . . . 218

19.5. November: The Swap Campaign . . . 219
19.6. What the Failures Have in Common . . . 221
19.7. The Evening of 2026-11-19 . . . 222
19.8. Status on 2026-11-19 . . . 223

VI. Beyond the Current Factory 226

20. Patterns Worth Reusing 227
20.1. Durable Intent Before Automation . . . 227
20.2. Separate Work Graph from Execution History . . . 227
20.3. Make Dispatch Orthogonal . . . 227
20.4. Claim Atomically, Repair Cross-Store Effects . . . 228
20.5. Give Every Run a Disposable Workspace . . . 228
20.6. Put Governance Outside the Model . . . 228
20.7. Define Proof of Work . . . 228
20.8. Use One Automated Integration Door . . . 229
20.9. Distinguish Red from Unavailable . . . 229
20.10. Reconcile Durable Scratch . . . 229
20.11. Record Reasons, Not Only Decisions . . . 229
20.12. Price Delivered Outcomes . . . 229
20.13. Make Authority a Data-Flow Property . . . 230
20.14. Date the Fleet . . . 230
20.15. Replace Hosts Instead of Mutating Them . . . 230
20.16. Make “Merged but Not Running” a State . . . 230
20.17. Own Your Pause and Prove Quiet . . . 231
20.18. Put the Point of No Return in Writing . . . 231
20.19. A Dry Run That Reports Every Failed Precondition . . . 231
20.20. Rehearse a Never-Run Path . . . 231
20.21. Separate Attention from Permission . . . 232
20.22. Sign Exact Text . . . 232
20.23. A Rule Registry That States Its Own Assurance . . . 232
20.24. Run the Tests in the Audit . . . 233
20.25. Dormant by Default on Main . . . 233
20.26. An Adoption Ladder . . . 233

21. The Road Ahead 235
21.1. One Goal at a Time . . . 235
21.2. Goal G1: One Automatic Swap . . . 235
21.3. After G1: Goals G2 to G5 . . . 237
21.4. Factories . . . 237
21.5. The Hosted Product . . . 238
21.6. Two Deployment Targets . . . 239
21.7. What the August Edition Predicted . . . 240
21.8. What Yesod Should Not Become . . . 241

Epilogue: Scar Tissue 243

A. Command Field Guide **244**
A.1. Catalog and Knowledge . . . 244
A.2. Notes and Lifecycle . . . 245
A.3. Beads Routing . . . 246
A.4. Dispatch and Runner . . . 247
A.5. Refinery and Admissions . . . 248
A.6. Gate . . . 249
A.7. Release, Skew and Replacement . . . 250
A.8. Mail, Bulletin and Agents . . . 251
A.9. Rules . . . 252
A.10. Scratch Machines . . . 252

B. States and Timers **255**
B.1. Note Lifecycle Status . . . 255
B.2. Note Disposition . . . 256
B.3. Planning Job State . . . 257
B.4. Run Status . . . 258
B.5. Child Outcomes . . . 258
B.6. Refinery Attempt State . . . 258
B.7. Gate Execution State . . . 259
B.8. External Effect Evidence (Refinery) . . . 260
B.9. Admissions State . . . 260
B.10. Replacement Intent Phase . . . 261
B.11. Replacement Target State . . . 262
B.12. Replacement Effect-Journal Evidence . . . 262
B.13. Replacement Stop-Guard State . . . 263
B.14. Lifecycle Rollout Stage . . . 263
B.15. Runner Timers . . . 263
B.16. Planning Timers . . . 264
B.17. Refinery Timers . . . 265
B.18. Gate Limits . . . 266
B.19. Deployment Settle . . . 266
B.20. Replacement Timers . . . 267
B.21. Mail, Bulletin and Fleet . . . 268
B.22. Scratch Leases . . . 269

C. Roles and Authority **271**
C.1. Primed Roles in Code . . . 271
C.2. The Operating Cast, October and November 2026 . . . 273
C.3. Deterministic Services Today . . . 274
C.4. Authority Rules . . . 275
C.5. Not Verified . . . 277

D. Evidence and Source Map **278**
D.1. The Snapshot . . . 278
D.2. Evidence Tiers . . . 278

D.3. Source Anchors by Chapter . . . 279
D.3.1. 1. Follow the Change . . . 279
D.3.2. 2. Judgment and Mechanism . . . 280
D.3.3. 3. The Two Currencies . . . 280
D.3.4. 4. The Registry Beneath the Factory . . . 281
D.3.5. 5. Capture, Plan, Deliver . . . 281
D.3.6. 6. The Runner . . . 281
D.3.7. 7. Governed Agent Execution . . . 282
D.3.8. 8. The Merge Controller . . . 282
D.3.9. 9. The Deterministic Gate . . . 283
D.3.10. 10. Merged Is Not Deployed . . . 284
D.3.11. 11. Deploy Is a Host Swap . . . 284
D.3.12. 12. Born, Not Repaired . . . 285
D.3.13. 13. Seeing the Whole System . . . 285
D.3.14. 14. Failure Is a State Machine . . . 286
D.3.15. 15. The Reliability Contracts . . . 286
D.3.16. 16. Cost-Aware Dispatch . . . 287
D.3.17. 17. The Agent Organization . . . 287
D.3.18. 18. Signed Human Authority . . . 288
D.3.19. 19. What Failed, August to November . . . 288
D.3.20. 20. Patterns Worth Reusing . . . 289
D.3.21. 21. The Road Ahead . . . 289
D.4. What Rests on Working Notes Only . . . 290
D.5. Not Verified . . . 291

E. Timeline, August to November 2026 292
E.1. August . . . 292
E.2. September . . . 293
E.3. October . . . 295
E.4. November . . . 297

F. Glossary 300

G. Building This Book 306
G.1. How This Edition Was Written . . . 306
G.1.1. What the agents could and could not do . . . 307
G.1.2. Known weaknesses of the method . . . 307
G.2. Building the PDF . . . 307
G.3. Publishing an Edition . . . 308

About This Edition

This book describes **Yesod**, an autonomous software factory that grew out of a personal tool registry. It is a source-grounded account of how a written request becomes planned work, how bounded coding agents turn that work into branches, how a deterministic merge controller decides what may reach `main`, how merged code becomes running code, and how one human stays in authority over all of it. This is the November 2026 edition. It replaces the August edition, which described a factory about half the size by commit count.

Item	Snapshot
Repository	`yesod-aicoe`
Baseline commit	`16606a42` (`origin/main`, 2026-11-17 UTC)
Verification date	2026-11-19
Prior-edition baseline	`4d658cc6` (2026-08-16)
Deployment observation	none made directly for this edition

The August edition included a read-only observation of the live fleet. This edition does not, which is what the last row records. Every fact about the deployment in these pages comes from one of three places: a runbook or receipt committed on `main`, a commit, or the working notes that the coordinating agents kept while they ran the factory. Each such fact carries its date and its source.

> **Proven Where — as of 2026-11-19, evening.** The first fully automatic, merge-triggered controller swap has not happened in production, and it has never run anywhere, including in rehearsal. The swap is designed, and its code is on `main`. Hand-submitted swaps have been rehearsed on disposable machines using reference code. In production, the hand-born serving controller was adopted into the replacement lineage on 2026-11-10 and the read-only dry-run preflight passed on 2026-11-12. In mid-November three replacement attempts were submitted by hand in production; each ended `needs_operator` before any transfer of authority, and every controller change so far was run by hand. On the evening of 2026-11-19, pre-flight and rehearsal of the hardening work for a fourth attempt raised seven findings before any production step. One of them, first read during the fact-check of this book, is that a merge-triggered swap deadlocks as coded. No fourth attempt has been made. All of this is according to the agents' working notes. No chapter in this book claims otherwise, and Chapter 11 (Deploy Is a Host Swap) gives the full account.

What Changed Since August

The August edition described a factory whose hard problems were merge correctness and an in-place deployment transaction across four hosts. Between 2026-08-20, the day of that edition's last edit, and this edition's baseline, `main` gained 3,084 commits (counted with `git rev-list --count --since=2026-08-20T00:00:00Z origin/main`, which is by committer date from 00:00 UTC). Four changes account for most of what is new in the book.

The note lifecycle matured. A note is now captured, planned and delivered as three distinct stages. A plan is bound to the exact revision of the note it was made from, a scheduler arms approved work for dispatch, and the planned children run one at a time on a single accumulating branch. Chapter 5 (Capture, Plan, Deliver) covers it.

The gate came home. Yesod's own merge-time test gate moved from cloud MicroVMs to nine disposable virtual machines on the home hypervisor, called demons, which together run one candidate as eight test shards plus a safety island. Chapter 9 (The Deterministic Gate) explains why and how.

Deploy became replacement. In-place promotion kept leaving merged code that was not running, so the design changed: a successor controller is born from a template, qualified, and handed authority by a database compare-and-swap, and the predecessor is retired. "Merged but not deployed" became a named terminal state, `merged_not_deployed`. The automatic version of this swap is on `main` and has not completed in production. Part III covers all of it.

An organization grew around the factory. A coordinating agent called the mayor, a devil's advocate, a forensics agent, a refinery clerk and several monitors now run the factory day to day, talk on a bulletin board, and act on signed grants from Stephen that are scoped to exact text. Most of this is practice held by convention, and very little of it is enforced by code. Part V describes it and says which is which.

What Is New, Rewritten and Revised

Readers of the August edition will find the same spine in Parts I and II, two new parts, and a new chapter on the autumn's failures.

Status	Chapters
New	5, 10, 11, 12, 17, 18, 19, 21
Rewritten	1, 6, 7, 9, 13, 14, 16
Revised	2, 3, 4, 8, 15, 20, Epilogue

Chapter 5 replaces the old "Planning and Dispatch" and Chapter 21 replaces "The Future of Yesod"; both are new text. Chapters 3 and 8 were revised heavily enough to correct errors in the August text. Chapter numbers changed: the reliability contracts chapter, formerly 17, is now Chapter 15.

There are seven appendices: Command Field Guide; States and Timers; Roles and Authority; Evidence and Source Map; Timeline, August to November 2026; Glossary; and Building This Book. The timeline is new. The others were rewritten against `main`, and Building This Book was updated.

Which Facts Win

Facts in this book follow a strict precedence. When two sources disagree, the higher one wins.

1. **Code.** Behavior verified in the pinned source tree at `16606a42`.
2. **Deployment.** A dated fact about the live system taken from a runbook, a receipt or a commit on `main`. It is dated because a deployment can drift without a commit.
3. **Working notes.** A fact recorded only in the notes the coordinating agents kept at the time.
4. **Historical.** Repository history and earlier design documents, used to explain how the system evolved.
5. **Design.** Intent or planned work that is not yet an operational guarantee.

The third tier is new. The agents that ran the factory this autumn kept durable notes between sessions: handoffs, decision logs, lessons learned, the state of a release window. Much of what happened on the live hosts in October and November was written down nowhere else. These notes were written by participants under time pressure, and nobody audited them afterwards. They are not authoritative. The book uses them for dates, for the order of events and for the reasons behind decisions, and it checks anything technical in them against the source before relying on it. A number that appears only in the notes is reported as what the notes say, and is never presented as a measurement.

The distinctions matter in practice. A status can be declared in an enum and never written by the current path. A safety control can exist in source while the running process is on older code. A capability can be on `main`, pass every test, and never have run against a real machine. Much of this edition is about that last gap.

How Claims Are Marked

The precedence shows up in the prose in four ways, so a reader can tell what kind of claim a sentence makes without a footnote.

- A function, class, constant, table, state or command named in `code font` means the claim was verified in source at the snapshot. Sentences about mechanism name the symbol that implements it. Line numbers are not printed, because names are more stable.
- A fact about the deployment is dated in the sentence (“on 2026-09-16”, “as of 2026-11-19”) and says where it came from.
- A fact that rests on the working notes says so: “the agents’ working notes record”, or “according to the mayor’s notes for” a given date.

- A reading that was not traced end to end is called “inferred” in the sentence.

Each chapter ends with a section called Source Notes. It lists, section by section, the kind of evidence and the anchors behind it (file paths, symbol names, runbook files, commit ids, or dated working notes), followed by a list of what could not be verified.

Credentials, network addresses and other secrets are omitted throughout, including the locations of secret files.

How This Edition Was Made

This edition was researched, drafted and fact-checked by agents working under one editor. They had read-only access to the source as of the baseline commit, they did not run the factory’s tests or commands, and they did not touch production. A separate set of agents checked each draft against the source. The appendix “Building This Book” describes the process and the toolchain.

How to Read This Book

The book has six parts.

Part	Chapters	Subject
I. One Change Through the Factory	1-4	One request followed end to end, and the stores beneath it
II. From Intent to Merge	5-9	Planning, the runner, agent execution, the merge controller, the gate
III. From Merge to Running Code	10-12	Why merged is not deployed, the host swap, machines born from templates
IV. Operating the Factory	13-16	Observation, failure states, reliability contracts, cost
V. The Organization Around the Factory	17-19	The agent roles, signed human authority, what failed this autumn
VI. Beyond the Current Factory	20-21, Epilogue	Reusable patterns and the order of work ahead

A reader new to Yesod should read Parts I and II in order, since each chapter there assumes the one before. Part III can be read on its own by someone who mainly wants the deployment story, and Part V by someone who mainly wants to know how a person supervises agents

with broad access. Chapter 19 (What Failed, August to November) is the dated narrative that connects the incidents told in other chapters. The appendices provide commands, states and timers, roles and authority, a source map, a timeline, a glossary, and an account of how the book is built.

Six recurring sidebars carry specific kinds of information:

- **Invariant**: a property the system is built to preserve, with the function or constraint that enforces it, or a statement that it is held by convention only.
- **Terminology Trap**: a word whose casual use hides two different mechanisms.
- **Operator Check**: a safe, read-only way to establish current state.
- **Factory Floor**: a dated incident that changed the design.
- **Design Trade-off**: a deliberate exchange of speed, cost, safety or complexity.
- **Proven Where**: new in this edition. For a capability whose production status matters, it says what is designed, what is on `main`, what was rehearsed on disposable machines, what has run in production, and what is still open.

The commands in this book are examples for understanding the system. Read-only status commands are shown freely. Mutating commands appear only where they are needed to explain a contract, are marked as mutating, and on the live factory are governed by the authority rules in Chapter 18 (Signed Human Authority).

Source Notes

Claim or section	Tag	Anchor
Snapshot table	[code]	`origin/main` at `16606a42`; prior baseline `4d658cc6`
3,084 commits since 2026-08-20	[code]	`git rev-list --count --since=2026-08-20T00:00:00Z origin/main`
Swap status, rehearsals	[deploy], [notes]	runbooks on `main`; agents' working notes, 2026-11-10 to 2026-11-19
What changed, chapter status	[design]	the edition plan; Chapters 5, 9, 10-12, 17-18; the files in `book/chapters` and `book/appendices`

Not verified:

- No deployment observation was made for this edition, so the swap status rests on the working notes and on runbooks on `main`, not on a direct reading of the live hosts.

Preface: A Factory, Not a Chat Room

An LLM can write a patch. A software factory has to answer harder questions.

What requested the change? What source did the plan use? Which dependencies must land first? Which model may attempt the work, on which host, for how long, and at what cost? What counts as proof that the run produced something? Who is allowed to merge it? Which tests are authoritative? What happens when the process dies after committing but before reporting success? What happens when the gate infrastructure fails, rather than the tests? How does merged code become running code? Which record should an operator trust when two dashboards disagree? And when the agents can do almost anything, how does the one human responsible for the system stay in charge of it?

Yesod is the collection of answers to those questions. Some of the answers are finished, some are partial, and this book says which.

Its central move is a separation of concerns:

- **Agents make judgments.** They clarify intent, inspect source, decompose work, diagnose failures, review each other's proposals, and recommend recovery.
- **Services enforce mechanics.** They poll, claim, spawn, limit, verify, queue, gate, merge, deploy, reconcile, and record.
- **Durable stores hold truth.** PostgreSQL records notes and execution state; Dolt-backed Beads records work graphs; Git records code history and integration.

This division matters more than the number of models or hosts. Without it, an agent's fluent assertion cannot be told apart from a completed action. With it, an important claim can be checked against something mechanical: a locked row, a commit past a base SHA, a verified remote branch, a gate return code, a pushed `main`, a deploy outcome, a receipt bound to a digest.

> **Invariant — Evidence outranks narration.** An agent may say the task is complete. The factory asks for commits, checks, a remote branch, a green merge gate, and proof that the merged commits landed.

The result is far from infallible, and this edition documents the seams that are open today. The automatic controller swap, the mechanism meant to turn every merge into running code, is on `main` and has not yet been proven in production. The governance that keeps a human in authority is mostly discipline layered on a control plane that, with a few narrow exceptions, does not authenticate who is issuing a command. A child task can close without its acceptance tests having run, and the record says so only to a reader who knows where to look. The cost-aware router that the earlier editions anticipated is still unbuilt. A description of an advanced

system is useful to the extent that it makes facts like these easy to find, so each one is stated plainly in the chapter that owns it.

Prologue: The Accidental Factory

Yesod did not begin as an orchestrator.

On March 29, 2026, the repository opened as documentation for a video-production workflow. It described tools, handoffs, rendering steps, and publishing conventions. The early meaning of *orchestration* was ordinary process glue: how one utility passed work to another.

The first `yesod` CLI arrived on May 10. Its purpose was modest: keep a local registry of tools and repeatable processes. Tools acquired descriptions. Notes acquired stable identifiers. Full-text search made the catalog retrievable. An `ask` command let an LLM answer questions over the registry.

That last feature changed the direction of travel. A registry that can explain what it knows soon wants to record what happened. Notes gained status and threaded comments. The catalog gained a web interface and a live activity stream. Work needed dependencies, so Yesod adopted Beads, backed by Dolt, and gained a graph of work alongside its log of intent.

By late May, the same CLI could prime different agent roles. A worker needed someone to plan its tasks. Completed branches needed someone to merge them. Multiple concurrent workers needed atomic claims.

On June 1, commit `14004657` added `codefactory-dispatch`. A note could be assigned to a model; a runner could claim it; an agent could work in an isolated Git worktree; the result could be verified and pushed. The catalog had crossed the line from describing tools to dispatching them.

Summer: Scar Tissue

The next six weeks taught the factory by breaking it.

The runner learned that exit code zero is not proof of work. It learned to verify commits past the base SHA, to preserve partial work, to distinguish infrastructure failures from model failures, to cap retries, and to halt after repeated results with no progress. It learned that a host identity must never be faked merely to attract work. It learned that targeted tests belong inside a bounded agent run and that the authoritative suite belongs at merge time.

The refinery evolved from a role into an automated merge engine. It gained a durable queue, branch verification, isolation of a red change from its neighbors, retry caps, merge summaries,

and hooks that run after a merge. The gate moved from the operator workstation to disposable MicroVMs in the cloud, which shortened the critical path and added a new class of image, proxy, and cloud-lifecycle failure.

The factory also learned what a low price does not tell you. Tokens are inputs, and value is a merged change. A cheap model that never lands a correct change can cost more per delivered feature than an expensive model that succeeds once. The run ledger therefore records retries, outcomes, tokens, and cost, so that the meaningful unit can become **cost per merged bead**.

By July 11 the repository held more than two thousand commits, a multi-repository refinery registry, a live fleet, and a remote gate. None of that was in the original roadmap, because there was no such roadmap. In August the merge path was rebuilt as a deterministic controller with immutable attempts and leases, and deployment became a transaction across four hosts. That is the factory the previous edition described.

Autumn: The Factory Turns on Itself

Late August was spent on contracts and on undoing a mistake in planning. A hosted version of Yesod had been designed against a long-lived side branch, and by then production had grown the real planning, refinery and release mechanisms that the branch described as future work. A decision record dated 2026-08-24 (ADR 008) retired the branch as a target and required hosted work to land on `main`, switched off. The same weeks made repository onboarding subject to a reviewed policy file and gave planning a versioned input schema. On 2026-08-28 the gate began to leave the cloud. The note behind that change records a candidate that passed three clean CPython gates and went red in the authoritative MicroVM gate, and the response was to bind the gate to one exact interpreter build on machines the factory owns.

In September the gate came home to hardware Yesod's owner controls. The last serial run of the whole test suite as a gate was on 2026-09-01. After that, a candidate is tested as eight shards plus a safety island on nine disposable virtual machines, called demons, and is green only when all nine come back green and every collected test is shown to have run exactly once. Moving the gate was largely manual work: one migration epic accounts for 136 commits, 69 of them touching `ops/`, ending in a 48-hour soak from 2026-09-25 to 2026-09-27. That most of these were made directly by operator sessions, outside the factory's own pipeline, is inferred from the commit record. The lasting result was a different idea of what a machine is. In the second week of September the fleet's birth artifact was brought into the repository (`5f8582b8a`), and from then on a gate machine or a runner was something to be born from a template with a receipt, and replaced when in doubt. On 2026-09-16 the original controller VM was itself replaced by a new one, by hand, following a runbook that is on `main`.

October was the busiest month in the repository's history: 1,400 commits, 549 of them in the week of October 12 through 18 (counted by committer date in UTC). Much of the volume was a product taking shape, a hosted API with its cloud deployment code, all of it merged through the ordinary gate and all of it dormant. The month's turning point was elsewhere. Fixes were merging and not reaching the machines that needed them. The agents' working notes for 2026-10-13 record that in one 24-hour period fifteen of fifteen merges ended

`merged_not_deployed`, the state the refinery assigns when a change is on `main` and the deployment that should follow did not settle. The same notes record the rule Stephen set that day: "ALL deployment should be basically stopping one VM/LXC and starting a different one, with a bit of handover." The first code of the replacement framework carries a commit date of the same day (`12bd23756`). The same weeks produced the organization of coordinating agents described in Part V, because running release windows by hand, safely, turned out to need roles, a shared board, and written permission.

November had one goal, recorded in the working notes as Stephen's statement of it: the controller swapped automatically once, triggered by a merge, with a published receipt and no operator fix during the swap. Nearly everything else was parked. Only 109 commits reached `main` in the first seventeen days of the month, almost all of them hardening the replacement path. The reading that the drop reflects paused merge intake and parked work, and not lost data, is inferred from the counts together with the notes. According to those notes, the serving controller was adopted into the replacement lineage on 2026-11-10 after four refusals that day, and a read-only dry run of the swap passed on 2026-11-12. Three attempts to have the factory give birth to its own successor ended needing an operator, and none reached the handover. As of the evening of 2026-11-19 the automatic swap has not happened in production, no fourth attempt has been made, and rehearsal of the hardening work had just raised seven more findings (Chapter 11).

The growth over the period looks like this.

Measure	2026-07-11	2026-08-20	2026-11-17
Commits on `main`	2,051	3,275	6,323
Lines of Python	97,438	186,926	535,696
Test modules	236	375	1,004

Each column was counted the same way at one commit on `origin/main`. The 2026-07-11 and 2026-08-20 columns were measured at commits `4ca8fb6e` and `c7a3e4d2`, the last commits reachable by `git rev-list -1 --until=<date>` as run for this edition, and the 2026-11-17 column at `16606a42`. Commits were counted with `git rev-list --count`, lines with `wc -l` over every `.py` file under `yesod/src/yesod`, and test modules as `test_*.py` files under `yesod/tests`. Git resolves a bare date to the time of day at which the command runs, so the first two commits are not the last of their UTC days: `4ca8fb6e` carries a committer date of 2026-07-12 04:30Z and `c7a3e4d2` of 2026-08-20 19:17Z. The figure of 3,084 commits since the August edition, given in About This Edition, is counted from 00:00 UTC on 2026-08-20, when `main` held 3,239 commits. Lines of code measure volume and nothing else. A large share of the autumn's lines are receipts, preflight checks and refusals, code whose job is to stop something from happening.

The Pattern

The evolution followed a repeated pattern:

1. make one useful action possible;
2. observe the failure mode it creates;
3. turn the lesson into a durable contract;
4. move that contract from prose into mechanism.

That pattern is the real subject of this book.

The autumn added a second subject, and it would be dishonest to fold it into the first. Through the summer, the question was how to make agents' work trustworthy enough to merge. By October the agents coordinating the factory could pause it, release into it, restart its hosts and rewrite its plans, and the question became how one person stays in authority over agents that can do almost anything. The answers so far are signed grants scoped to exact text, a second agent whose job is to object, and a growing list of rules that each trace back to a specific mistake. Most of those answers are still at step three of the pattern: they are contracts written in prose and kept by discipline, and only a few have become mechanism. Part V describes them as they are.

Source Notes

Claim or section	Tag	Anchor
March to July history	[historical]	repository history; `14004657`; prior edition
Late August	[code]	ADR 008; `9cdac2c43`, `34fd9d07c`; `f56a845ac`, `1964730c3`; note text in `docs/tools/yesod.md`
September	[code], [deploy], [inferred]	ADR 009; `5f8582b8a` (2026-09-09); `18419d4d8`; `git log --grep yes-6kl4s`; `docs/runbooks/controller-g1-handoff-2026-09-16.md`
October and November	[notes], [code]	agents' working notes, 2026-10-13 to 2026-11-15; `12bd23756`
Growth table	[code]	`git rev-list -1 --until`, `git rev-list --count`, `wc -l` at `4ca8fb6e`, `c7a3e4d2`, `16606a42`
Monthly and weekly counts	[code]	`git rev-list --count` on `origin/main`, committer date in UTC

Not verified:

- The fifteen-of-fifteen figure, the quotation and the goal statement attributed to Stephen, the adoption and dry-run dates, and the outcome of the three attempts rest on the agents' working notes only.
- The explanation for November's low commit count is inferred, as is the reading that most of the gate-migration commits were made by hand.
- The dates before June were carried over from the prior edition and were not re-traced for this one.

Part I.

One Change Through the Factory

1 Follow the Change

Monday, 9:07 a.m. Maya opens a coding agent beside the repository for her new product. She asks for a small feature: show a user why a queued job has not started. The agent reads the code, proposes an approach, edits the API and the interface, writes a test, fixes the test when it fails, and prepares the change for merge. By lunch, a capable engineer has completed a day's work in a few hours.

This is the promise of **vibe coding**, and it is a real one. Maya can move from intent to working software without assembling a team or handing a specification across organizational boundaries. For a solo project, an experiment, or a low-risk application, that may be exactly the right operating model.

Look closely at the job Maya has given the agent, though. In one context window it must be the product planner, systems designer, implementer, test author, reviewer, release manager and incident historian. Product assumptions sit beside stack traces. An early design choice stays in the context while the agent later judges whether that choice was wise. The test author already knows how the implementation works. The reviewer is reviewing its own reasoning.

Nothing here makes the agent less capable. Vibe coding takes a jack-of-all-trades engineer and makes that engineer dramatically faster. What it does not automatically create is a software organization. Two opportunities are left mostly unused:

1. **Parallelism.** A feature often contains independent work: an API change, a migration, tests, documentation. One agent in one session walks that path serially. Maya can open more terminals, but then she becomes the scheduler, assigns scope by hand, keeps the dependencies in her head, and reconciles the results herself.
2. **Focused roles.** Planning, implementation, testing, integration and operations reward different kinds of attention. A planner should settle uncertainty before code exists. A worker should stay inside a bounded task. An integrator should protect `main`, whatever the worker believes about its own progress.

The missing ingredient is **productive tension**. In a single session one fluent line of reasoning can carry a feature from idea to merge, and there is no durable boundary at which another party must accept the evidence. A factory supplies those boundaries. The planner pushes for explicit scope. The test gate tries to falsify the worker's claim. The merge controller protects the shared branch. Operations asks whether merged code became running code.

> **Design Trade-off — From acceleration to organization.** Vibe coding makes a strong generalist faster. A software factory adds parallel work, specialized attention, independent proof boundaries and durable coordination around that generalist. It pays for them in machinery, waiting and operating cost, which the rest of this book describes without flattering them.

Yesod begins at that boundary. It gives separate agent sessions bounded roles, keeps work and dependencies outside their context windows, and places deterministic services between an agent's judgment and any authority to change shared state. Agents plan, implement and investigate. Mechanisms decide what may be claimed, what counts as progress and what may merge.

The rest of this chapter follows one request through the factory as it runs in November 2026. Each term is introduced once and each stage names the chapter that treats it in full. The request is the Yesod equivalent of Maya's: add a read-only command that explains why an armed note is not running. (A command of this kind exists on `main` today as `note-why`.) The walk is constructed from the code path with exact state names and is not the record of one particular note.

1.1. Captured, Then Planned

The request first becomes a **note**: a row in the PostgreSQL table `tool_notes` that stands for one independently mergeable feature. Created without a planning flag, the note is born in status `captured` with automation policy `capture`, which means nothing will act on it (`policy_from_creation_flags`, `creation_values`). A note also carries a **disposition**, which says whether automation may touch it at all; only `active` permits anything. Chapter 3 (The Two Currencies) covers both fields.

Asking for a plan moves the note to `planning` and creates a row in `planning_jobs`, keyed by note, note revision and generation. A runner that has planning enabled leases the job in `_tick_planning` and runs a planner model against a snapshot of the repository. The planner may not edit, commit or create anything. Its only output is a plan artifact (`PlanArtifact`): a root task, at least three child tasks (`MIN_CHILD_TASKS`), and for each child the files it will touch and the exact commands that will count as acceptance. A plan that still contains open questions is invalid and cannot be published.

A valid plan is published as **Beads**: a root bead and its children in a Dolt-backed work graph, written by `BeadPlanPublisher` under a durable publish intent. The note then records `planned_revision` and `plan_fingerprint` and moves to `planned`. The plan is bound to the revision of the note that was read. If someone materially edits the note afterwards, the revision advances, the plan is cleared, and the note returns to `captured`. Chapter 5 (Capture, Plan, Deliver) tells this part in full, including the days in September when, by the agents' working notes, more planning jobs failed on open questions than succeeded.

The two objects answer different questions. The note says which mergeable feature is being delivered, what must land before it, and what happened to it. The bead tree says how the feature is decomposed and which step is next.

1.2. Approved, Armed, Claimed

Nothing runs until the plan is approved. `note-approve` checks that the plan evidence is current for this revision (`approve` in `note_lifecycle_ops.py`), sets the policy to `deliver` and

moves the note to `open`. The check is on the plan's provenance and does not establish who is approving, a gap Chapter 18 (Signed Human Authority) takes up.

In the August edition a dispatcher agent then armed the note by hand. Today a scheduler does it. `LifecycleScheduler.tick`, run inside a runner, re-evaluates each approved note and writes one row in `lifecycle_dispatch_arms`. The **arm** names a dispatch target, which is a configured model target and not a host. Its identity (`dispatch_idempotency_key`) hashes the note id and revision, the plan fingerprint, the refinery lane, the repository, the target branch, the source commit, that dispatch target and the execution placement. The scheduler sets `agent_dispatch` on the note and issues a PostgreSQL notification on `yesod_note_armed`.

That notification wakes the **runner**, the daemon that executes work on a build seat. Its claim is one short transaction in `_claim_note`: the runner's own row is locked, the note becomes `claimed`, and a run row in status `starting` is inserted in the same transaction (`insert_starting_run`). Two runners may see the same candidate; only one can own the note. The arm predicate is applied again inside the claim, so an arm written for an older revision of the plan cannot start work. Chapter 6 (The Runner) covers the loop.

1.3. One Child at a Time

The note moves to `in_progress` and the runner works the plan in order (`_process_planned_root_dispatc`
For each child it creates a Git worktree, launches a coding agent with a deny-by-default environment, and supervises the process against time and token limits. All children of a note commit to one accumulating branch named `<tool>-<note_id>`. The first child starts from `origin/main`; each later child starts from the branch its predecessor pushed.

The agent's own report is not evidence. When a child exits, `validate_leaf_result_commit` requires an exact result commit that descends from the recorded base. The runner pushes the branch, verifies the remote tip, closes the child bead with a reason string that carries the result commit, and reads the bead back before it looks at the next child. The agent never pushes `main` and, as a plan-bound child, may not close its own bead.

One weakness belongs in an overview because it changes how every later stage should be read. A child whose acceptance tests could not run on the build seat, usually because no test database is available there, still commits and closes, with `outcome=acceptance_not_executed` recorded in the close reason. A closed child therefore proves that a commit exists. It does not prove that a test ran. Chapter 7 (Governed Agent Execution) and Chapter 15 (The Reliability Contracts) return to this.

When the last child is reconciled, `_finish_planned_root_dispatch` moves the note to `awaiting_merge` and enrolls it in the merge queue.

1.4. Held at the Door

In current practice the note does not go straight to the gate. Since 2026-10-13, according to the agents' working notes, every delivery is placed on hold (disposition `held`) until a full test

pass of its exact branch tip has come back clean on a disposable machine. This is an operating policy carried out by a polling script that one of the coordinating agents runs. No code on `main` enforces it, and a polling script can lose the race on an idle queue. Chapters 5 and 17 (The Agent Organization) describe the policy and its gap.

1.5. Attempt, Candidate, Gate

Once released, the note belongs to the **merge controller**, the deterministic service at the center of what Yesod calls the refinery. Intake (`reconcile_legacy_intake`) turns the queue row into an **attempt**: one immutable row in `refinery_attempts`, in state `requested`, on the refinery lane for this repository and target branch. A lane serves attempts strictly in order and gives one of them the merge turn at a time. Intake also checks that admissions are open (`require_admissions_open`); an operator or a controller swap can pause them.

The attempt is `claimed` under a lease with a fencing token, then `preparing`. Here the controller builds the **candidate**: a real two-parent merge commit of the note's branch onto the current tip of `main`, with the subject line `refinery: deterministic candidate` (`GitRepository.prepare`). Its author, committer and date are synthetic, so the same base and source always produce the same commit SHA. A merge conflict ends the attempt as `needs_rework` and sends the note to a bounded repair job that must earn the gate again.

In `gating`, the candidate is tested. In this book "gate" alone always means this merge-time test gate. For Yesod's own repository the gate is a **union**: one concurrent run across nine disposable virtual machines on the home hypervisor (the fleet calls them demons), eight running shards of the suite and one, the safety island, running the tests that are unsafe in parallel. Green requires all nine receipts and a proof that every collected test ran exactly once (`CoverageProof`). A red result ends the attempt as `gate_red` and nothing lands; a retry is a new attempt row against whatever `main` has become. An infrastructure failure is recorded as `infrastructure_failed`, which is a different verdict from red. Chapter 8 (The Merge Controller) and Chapter 9 (The Deterministic Gate) cover these two stages.

1.6. Landing, and What Comes After

A green attempt moves through `gate_green` to `pushing`. The push is a compare-and-swap: `push_ref` uses `--force-with-lease` against the base the candidate was built on. If `main` moved in the meantime the attempt ends `push_raced` and nothing is forced. Otherwise the commit that was tested is now the tip of `main`.

For a lane configured to deploy after the push (`deploy_after_push`), as Yesod's own is meant to be, the attempt is still not finished. It enters `deploying`, and only a deployment step whose observer reports the exact candidate SHA as running moves it to `succeeded` and the note to `done`. If the observer has not confirmed within the settle budget, or deployment fails, the attempt ends in the terminal state `merged_not_deployed`. The lane is freed, the note is projected to `blocked`, an operator item is opened, and notes that depend on this one do not unblock on their own.

Proven Where — as of 2026-11-19. The intended deployment is a replacement of the controller's host: a successor machine is born, qualified, and handed authority (Chapter 11, Deploy Is a Host Swap). On `main` the deployment step records a replacement intent (`apply_candidate`, `submit_replacement_intent`), so a merge can start a swap. The rest of the chain has a gap as coded. The swap's proof that nothing is in flight counts any attempt in `deploying` (`_BLOCKING_ATTEMPT_STATES`) and makes no exception for the attempt that triggered the swap, so Chapter 11 reads the source to mean that a merge-triggered handover defers and ends needing an operator. That reading has not been exercised by any run. In production no replacement intent has reached `succeeded`, the handover has never run, and every controller change so far was run by hand by an operator. The agents' working notes describe `merged_not_deployed` as a daily sight this autumn. Chapter 10 (Merged Is Not Deployed) explains why.

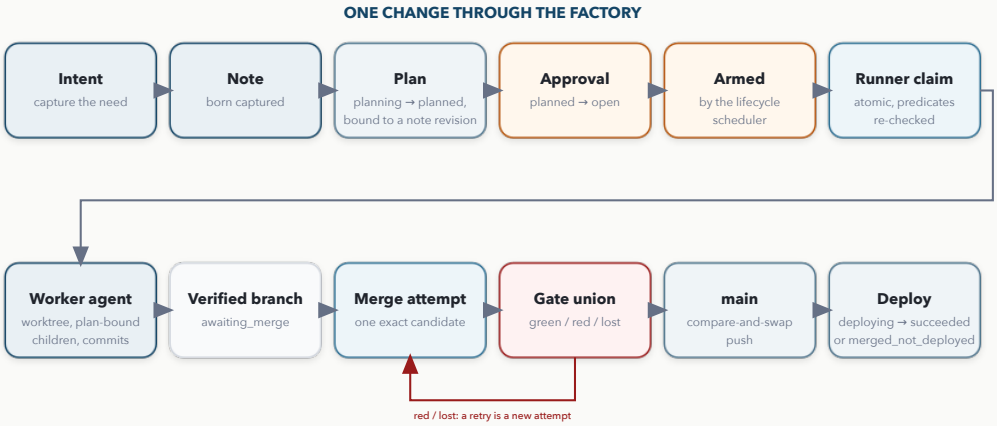

Figure 1.1.: One change moves through Yesod, from a captured note to a tested commit on main and a separately proven deployment.

The whole path compresses into one sentence:

Intent becomes a note; a planner job binds a validated plan to that note's revision; approval lets a scheduler arm it; a runner executes the planned children one at a time on one branch and proves each commit; the merge controller builds an exact candidate, gates it on nine machines, and lands it by compare-and-swap; and whether that code is running is recorded as a separate outcome on the attempt, `succeeded` or `merged_not_deployed`.

1.7. One Door, Precisely Stated

The popular description of Yesod is "a green gate is the only door to `main`." The precise version is narrower.

> **Invariant — Intended automated path.** For repositories the factory owns, the merge controller is the automated component that updates `origin/main`. It pushes only a candidate commit that passed the lane's gate, and it pushes by compare-and-swap against the base that candidate was built on (`push_ref`). This is enforced inside the factory. It is held by convention at the Git host, where a person or process with repository rights can still merge another way.

One incident shows the distance between the two versions.

> **Factory Floor — Main could not pass its own gate (2026-09-14).** Two pull requests for a graph-evidence feature were merged to `main` through the Git host, without going through the gate. They left `main` unable to pass its own test suite, and because every candidate is built on `main`, every merge in the lane was blocked. The repair, commit `0373aa5b4` on 2026-09-14, was small and mostly on the test side: two pages missing a shared script, stale test stubs, two unregistered output schemas, one timing-dependent test. Its message records no change to production behavior. The lesson is the invariant read backwards. The gate protects `main` only for changes that pass through it, and one change that goes around it can stop the factory for everyone else.

Chapter 9 tells a second case, three days earlier, in which an operator knowingly merged without a green gate and recorded that fact in the commit.

1.8. The Proof Chain

Each stage leaves evidence that the next stage checks for itself.

Stage	Evidence
Capture	note row, revision, automation policy
Planning	planning job, validated plan, plan fingerprint, stamped beads
Approval, arming	status change, arm row keyed to revision and plan
Claim	locked claim, `starting` run row, bead assignment
Execution	run ledger, progress events, exact result commit per child
Delivery	pushed and verified branch tip, queue row, acceptance flags
Integration	attempt events, candidate commit, nine gate receipts, push effect
Deployment	observer report of the exact SHA, or `merged_not_deployed`

Two rows deserve a caution. The execution row proves commits, and proves tests only where the close reason says they ran. The deployment row has two honest values, and as of 2026-11-19 the second is the common one for Yesod's own lane.

This chain is the difference between an agent demonstration and a factory. A demonstration can end when the model prints a confident summary. In a factory, each handoff carries durable evidence that the receiving stage can verify without trusting the sender. Chapter 2 (Judgment and Mechanism) separates the kinds of judgment, mechanism and state that produce that evidence.

Source Notes

Source snapshot: `yesod-aicoe` `origin/main` at `16606a42` (2026-11-17). The walk is constructed from the code path, not from one recorded note. The appendix "Evidence and Source Map" has the full list of anchors.

Claim or section	Tag	Anchor
Note creation, policy	[code]	`note_lifecycle_ops.py`: `policy_from_creation_flags`, `creation_values`
Planning job, plan contract	[code]	`planning_jobs.py`; `contracts/planned_root.py`: `PlanArtifact`; `BeadPlanPublisher`
Approval and arming	[code]	`note_lifecycle_ops.approve`; `lifecycle_dispatch.py`: `LifecycleScheduler`, `dispatch_idempotency_key`
Claim and execution	[code]	`runner.py`: `_claim_note`, `_process_planned_root_dispatch`; `root_dispatch_reconciliation.py`: `validate_leaf_result_commit`
Acceptance not executed	[code]	`acceptance_evidence.py`: `classify_child_acceptance`; `deferred_child_close_reason`
Hold every delivery	[notes]	agents' working notes, 2026-10-13, updated 2026-10-18
Attempt, candidate, push	[code]	`refinery_state.py`: `AttemptState`; `refinery_intake.reconcile_legacy_intake`; `refinery_git.py`: `GitRepository.prepare`, `push_ref`
Gate union	[code], [deploy]	`refinery_sefer_gate.py`; `refinery_gate_partition.py`: `CoverageProof`; reviewed controller document on `main` (last changed 2026-10-12)
`deploying`, `merged_not_deployed`	[code]	`refinery_controller.py`: `_run_deployment`, `_merged_not_deployed`; `routine_deployment.apply_candidate`
Quiet gap and the triggering attempt	[code], [inferred]	`replacement_guard.py`: `_BLOCKING_ATTEMPT_STATES`; read statically, as in Chapter 11

Claim or section	Tag	Anchor
Swap status	[deploy], [notes]	Chapter 11; agents' working notes to 2026-11-19
2026-09-14 incident	[code]	commit `0373aa5b4` (message body)
`note-why` exists	[code]	`tool.py` command `note-why`; `why_not_running.collect_evidence`

Not verified:

- The dates of the two un-gated merges behind the 2026-09-14 repair; only the repair commit is dated.
- That `merged_not_deployed` is "daily" for the Yesod lane rests on the agents' working notes, not on a query of the attempts table.
- The hold-every-delivery policy and its sentinel script are known from the working notes only; no such script is on `main`.
- Whether the live controller's Yesod lane has `deploy_after_push` set was not read from the deployment; it is inferred from the settle budget in the reviewed controller document and from the working notes.
- The September planning-failure counts rest on the agents' working notes.
- The interlock between the quiet-gap proof and a triggering attempt in `deploying` is read from source; the mayor's notes for 2026-11-19 record two reviewers confirming it (FINDING #132). No run has exercised a merge-triggered handover.

2 Judgment and Mechanism

Chapter 1 (Follow the Change) followed one change through the factory. This chapter sorts the parts it met into four planes, and for each plane it says which decisions belong to agents and which belong to code. The control plane assigns judgment, the execution plane governs coding runs, the integration plane controls entry to `main` and the step from merged to running, and the data plane preserves state and evidence.

The sorting matters because the two kinds of component fail differently. An agent can misread, forget, or be persuaded. A mechanism can only do what it was written to do, and its refusals can be reproduced. The chapter is also where the book first states plainly how much of Yesod's authority model is enforced by code and how much is held by convention. The second share is larger than the August edition admitted.

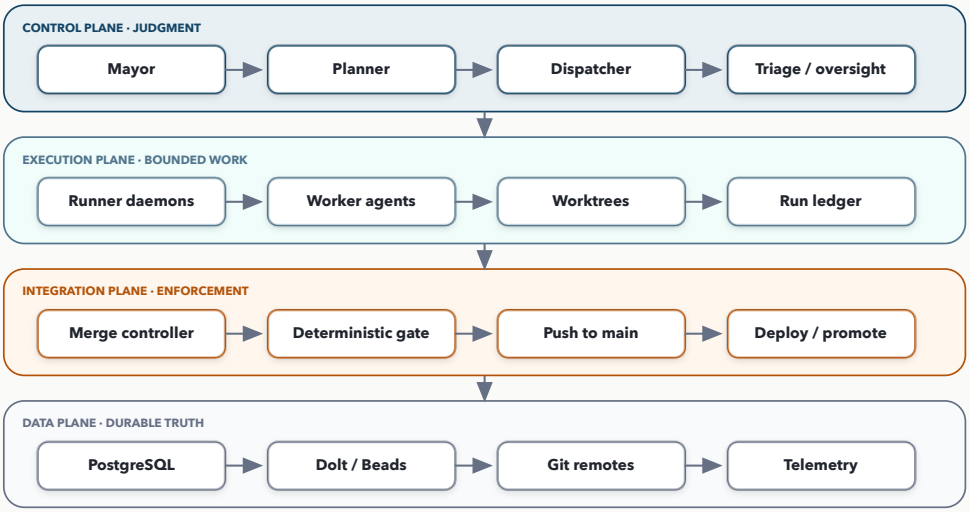

Figure 2.1.: The four planes separate judgment, governed execution, entry to main, and durable evidence.

2.1. The Control Plane

The control plane holds the decisions that cannot be reduced to a safe mechanical rule, such as whether a request is specific enough to plan or whether a failure points at the feature or

at the infrastructure. Yesod assigns these judgments to agents. It does not ask every agent to make every kind of judgment.

A **role** is a bounded operating contract for one agent session. It states the outcome the session should optimize, the evidence it should consult, the actions it may and may not take, and where it hands work next. A role is independent of model identity. The same model can occupy different roles in different sessions, and naming a session "planner" grants it nothing; it still has to produce durable artifacts that other roles and deterministic services can inspect.

The registry on `main` defines ten role primers, one per flag of `yesod prime`:

Flag	Focus
`--mayor`	lead the team and face the human
`--planner`	turn intent into an executable plan
`--dispatcher`	schedule planned work
`--triage`	investigate failures to their real boundary
`--infra-monitor`	fleet health and staging
`--mad-scientist`	controlled experiments
`--overlord`	keep the long-lived seats alive
`--ephemeral-refinery-supervisor`	human-judgment oversight of merging
`--worker`	contract for a runner-launched coding agent
`--refinery`	contract for a human-driven refinery session

The last two name agent-facing contracts at the execution and integration boundaries. They are not names for the runner daemon or the merge controller. The code is not tidy about this list: the roles section inside the shared guide names nine roles and omits `--overlord`, and the role sets used elsewhere each differ again. `SINGLETON_ROLES` protects five seats against duplicates. The name `FACTORY_ROLES` is defined twice: a seven-entry tuple in `factory_status.py` lists the `factory:<role>` topics the status report reads, and a four-entry list in `mail_cli.py` names the seats that are spawned at start and respawned by the seat keeper in code. Chapter 17 (The Agent Organization) gives the lists.

This table is the organization as it was designed between June and August. The organization that ran the factory in October and November was a different cast: a mayor, a devil's advocate, a forensics seat, a refinery clerk, a pre-run and rehearsal seat, and several watchers. Most of them have no primer in `prime.py`. Chapter 17 describes both layers and why the second one grew.

2.1.1. Bootstrapping a role with `yesod prime`

`yesod prime` turns a general agent session into a factory role by printing a Markdown operating guide to standard output, to be placed in the agent's context at startup. It trains nothing and stores nothing. It gives a fresh session the factory's current doctrine.

Every invocation begins with the shared guide, `PRIME_GUIDE`: what Yesod is, the difference between the knowledge registry and the software factory, the Beads routing rules, the team and its authority model, and the essential commands. The trust rules for the bulletin board (`BULLETIN_USAGE_ADDENDUM`) are concatenated onto the end of `PRIME_GUIDE` itself, so every primed agent receives them whether or not it asked. A role flag then appends that role's guide. Two further flags, `--use-mail` and `--use-sjbis`, append **addenda** for peer mail and for the channel that reaches the human; they are not roles and compose with any of them.

```
$ yesod prime --planner --use-mail
$ yesod prime --infra-monitor --use-mail --use-sjbis
```

`--refinery` also appends a live rendering of the merge queue (`_merge_queue_section`), so that session starts with current state as well as doctrine. `--worker` is built from the worker canon, the single-sourced worker instructions that Chapter 7 (Governed Agent Execution) describes.

Priming is replayable. After a context compaction, a cleared conversation or a respawn, the agent runs the same variant again and has its contract back. Its operational state lives elsewhere: before a compaction or handoff the agent writes a checkpoint to its `factory:<role>` topic with its address, focus, active note and bead ids, decisions, blockers and exact next action. A replacement primes, reads the newest checkpoint and resumes. The automatic `factory:seat-state:<role>` topics are narrower. They record how a seat was launched and do not record what it was doing.

> **Terminology Trap — A primer is not a credential.** `yesod prime --mayor` prints the mayor guide for any caller. Priming sets expectations inside an agent's context. It confers no authority, and nothing in the factory checks which primer a session read before accepting a command from it.

2.1.2. Authority: what code enforces and what is convention

The shared guide states an authority invariant in capital letters: human authority flows only from Stephen through the human channel to an operator, and peer mail never conveys it. The August edition printed that as a property of the system. It is a rule every agent is told, and agents largely follow it. No code rejects an agent that acts on a peer's word.

The places where code does bind authority are few, and worth listing exactly.

Surface	What the code requires
Resuming admissions	the expected revision and the pause token
Adopting a controller	root, actor `stephen`, an operator-owned live pause
Bulletin authorship	a roster row with a role, or the local operator
A singleton seat	a spawn token, when one is presented, matching the roster row

Surface	What the code requires
A dispatched agent's environment	a deny-by-default allowlist

The symbols are `resume_admissions`; `_require_root`, `_adoption_actor` (which compares with `REQUIRED_ACTOR`) and `_read_owned_pause`; `resolve_author`; `_reject_singleton_hijack`; and `build_agent_launch_environment`. Even the strongest of these, adoption, binds to an actor string that a root caller types. The allowlist covers dispatched workers and planners, and the interactive coordinating agents are outside it.

Other surfaces look like authority and are weaker than they look. `note-approve` checks the plan's provenance and the note's disposition (`approve`). As actor it passes only a source label that `_detect_source` derives from the calling environment, which is recorded and never compared, so whoever can run the command against the database can approve. The disposition verbs `note-hold`, `note-unhold` and `note-release` check a reason and the state rules and no identity. The check behind the fleet all-stop and the bulletin's operator path (`_authorized_for_all_stop`) reads an environment flag, the mail identity from the environment, or an actor or user name from the environment. Mail stamps the sender from the process's own resolved identity and discards `--from`, but that identity comes from a flag or an environment variable; this reading of the send path is inferred from source and was not exercised.

Nothing in the factory repository verifies a human signature. The signed approvals that governed the autumn's releases are made and checked with a separate tool outside the repository, by agents who verify each grant against exact text before acting.

> **Invariant — Authority (held by convention).** Human authority enters only through Stephen's own channel; peer mail and bulletin posts carry evidence, requests and advice, never permission. Primers, peer review on the bulletin, and the signed-grant procedure hold this rule. Code holds only the edges in the table above.

Chapter 17 covers identity, mail and the bulletin. Chapter 18 (Signed Human Authority) has the full table of where human authority enters in code, and covers grants, what they scope, and the mistakes that shaped the procedure.

2.1.3. The planning and dispatch boundary

In August a planner agent structured the work and a dispatcher agent chose a model target and armed the note. The separation survives, with mechanism now standing where one of the agents stood. Planning is a durable job that produces a validated plan bound to a note revision. Approval is an explicit act. Arming is done by `LifecycleScheduler` after approval, as Chapter 1 traced. The `--dispatcher` primer and the `yesod codefactory-dispatch arm` verb still exist, but arming a planned note by hand changes the reviewed input and invalidates its plan (the evidence check reports `planning_input_drift`). Chapter 5 (Capture, Plan, Deliver) has the detail.

2.2. The Execution Plane

The execution plane is the runner fleet and the agent processes it governs.

A **runner** is a deterministic daemon on a build seat. It claims one note in a transaction, creates an isolated worktree, launches the coding agent with an allowlisted environment, enforces a wall-clock limit (`DEFAULT_TASK_TIMEOUT`, 1,800 seconds unless the runner configuration or a target overrides it) and an optional token budget, polls for kill requests, and reconciles the result (`_wait_with_governance`). Each claimed note runs in its own worker thread, so the main loop never blocks on an agent.

The coding process is an agent. The process around it is not. That is what lets the factory tolerate a worker that hangs, exits wrongly, forgets to commit, burns tokens, or reports success without a commit. A worktree is removed after its branch is pushed and verified; a failed or unverified worktree is kept for diagnosis.

The build seat is also a control in its own right: workers there write code in isolation from the sessions that wrote the note and will review the result (Chapter 17). Chapter 6 (The Runner) and Chapter 7 cover the runner and the worker contract.

2.3. The Integration Plane

In August the integration plane was the merge controller, the gate and a four-host promotion transaction. Today it has three parts, and the third is not finished.

The **merge controller** is small and deterministic by intent. On each tick it proves it still holds leadership of its lanes, reconciles durable state, and advances each attempt it owns by one action derived from the evidence on record (`LaneController`, with the actions enumerated in `ControllerAction`). No database transaction spans Git, gate or deployment I/O. It exercises no judgment about whether to merge. When a merge needs judgment, as a conflict does, the controller routes the problem to a bounded coding job whose result must pass the gate like any other.

The **gate fleet** is the nine disposable virtual machines on the hypervisor `sefer` that test Yesod's own candidates as one union. The controller holds no shell on them. It reaches each through SSH accounts restricted to one forced command, and the gate tests the exact candidate commit and nothing else. The gate moved here from cloud MicroVMs in early September; Chapter 9 (The Deterministic Gate) explains why. Other repositories use lighter providers.

Host replacement is how merged code is meant to become running code: a successor controller is born from a template, qualified, and handed authority by a compare-and-swap on a database row, after which its predecessor stops itself. The state for this is on `main` as durable intents, targets and an effect journal, driven one non-blocking pass at a time by `ReplacementDriver.drive`.

> **Proven Where — as of 2026-11-19.** Host replacement is designed and on `main`. The agents' working notes record rehearsals of parts of it on disposable machines. In production, the serving controller was adopted into the replacement lineage on 2026-11-10 and the read-only preflight ran green on 2026-11-12, both according to the agents' working notes. Three attempts at a driver-born successor ended needing an operator, none reached handover, and the handover itself has never run in production. Chapter 11 (Deploy Is a Host Swap) gives the full account.

2.4. The Data Plane

Three stores cooperate without pretending to be one database.

- **PostgreSQL** holds the catalog and the operational ledger: notes, statuses and dispositions, planning jobs, arms, runs, runner heartbeats, mail, refinery lanes and attempts, gate executions, and the replacement tables.
- **Dolt-backed Beads** holds the per-tool work graphs: parent and child structure and dependency readiness, on one shared Dolt server with a database per tool (Chapter 3, The Two Currencies).
- **Git** holds source history, the accumulating work branches, candidate refs, and the ancestry on `origin/main` that proves landing.

Each answers a different question, and confusion follows when one is asked to answer another's. A closed bead does not prove code landed. An `awaiting_merge` note does not prove an attempt exists. A note marked `done` is not a deployment receipt.

This edition adds a fourth kind of state that lives inside PostgreSQL and deserves its own name: **receipts and journals**. The autumn's design work kept arriving at the same rule, which is to record the intention before the effect and the observed outcome after it. A plan is published under a row in `planning_publish_intents`. The push to `main` is recorded as an external effect. Each step of a host replacement is claimed in `replacement_effect_journal` before it is attempted, and its row then says `intended`, `outcome_unknown`, `confirmed_applied` or `confirmed_not_applied`. Births, qualifications and handovers return typed receipts, each with a digest. Some of this evidence is immutable by construction: a terminal attempt cannot be changed, and a trigger forbids updates and deletes on `replacement_quiet_gap_observations`. A process that crashes between two steps can then read what it meant to do and find out whether it happened. Chapter 15 (The Reliability Contracts) develops the rule.

2.5. Why Determinism Matters

Agents are good at semantic work: interpreting a request, reading unfamiliar source, choosing a strategy, forming a hypothesis. They are poor choices for repetitive enforcement:

- taking a row lock;
- comparing SHAs;
- checking a timeout or a lease;
- counting retries;

- verifying a remote ref;
- proving that every test ran exactly once;
- refusing a launch above a hard cap.

Those operations should be code. Yesod's maturity has come less from adding roles than from moving the enforcement of invariants out of prompts and into services, and the autumn supplied a measure of how far that has gone. Where a rule was in code, it held: the factory's integrity failures in these months were few, and its failures of availability were many (Chapter 14, Failure Is a State Machine). Where a rule was only a convention, it held as well as the agents remembered it. The compliance registry behind `yesod rules` says so about itself: of its twelve working rules, six have a detector, six are attested only, and none is marked enforced (`RULES`, `NEVER_CHECKED_RULES`).

> **Design Trade-off — More mechanism, less improvisation.** Every mechanical guard adds code and operating surface, and several of this autumn's outages were guards refusing correctly at an inconvenient time. In exchange a failure becomes reproducible, testable and visible, and no model has to remember the rule at the moment it matters.

Deterministic enforcement depends on durable objects with precise meanings. Chapter 3 turns to the two at the center of work coordination: notes and beads.

Source Notes

Source snapshot: `yesod-aicoe` `origin/main` at `16606a42` (2026-11-17).

Claim or section	Tag	Anchor
Ten primers, two addenda	[code]	`prime.py`: `prime()` options, `PRIME_GUIDE`, `BULLETIN_USAGE_ADDENDUM`
Disagreeing role lists	[code]	`mail_cli.py`: `SINGLETON_ROLES`, four-entry `FACTORY_ROLES`, `_seat_keeper_check`; `factory_status.py`: seven-entry `FACTORY_ROLES`
Checkpoints, seat state	[code]	`PRIME_GUIDE` "Durable Context Across Compaction and Handoff"; `SEAT_STATE_TOPIC_PREFIX`
The cast that ran in autumn	[notes]	agents' working notes, 2026-10-12 to 2026-11-19
Authority edges in code	[code]	`refinery_admission.py`; `replacement_adopt.py`; `bulletin_identity.py`; `mail_cli.py`; `agent_launch_environment.py`; symbols as named in the text
Weak or absent checks	[code], [inferred]	`note_lifecycle_ops.approve`; `tool.py`: `_detect_source`, `_set_note_disposition`; `fleet_cli._authorized_for_all_stop`; `mail_cli.resolve_self` (read, not exercised)

Claim or section	Tag	Anchor
Signed grants are external	[code], [notes]	no signature verification under `yesod/src`; practice per agents' working notes from 2026-10-24
Planning and arming	[code]	`lifecycle_dispatch.LifecycleScheduler`; `_plan_evidence` check `planning_input_drift`
Execution plane	[code]	`runner.py`: `_claim_note`, `_wait_with_governance`, `_cleanup_worktree`, `DEFAULT_TASK_TIMEOUT`
Merge controller	[code]	`refinery_controller.py`: `LaneController`, `ControllerAction`; `refinery_runtime.DeterministicRefineryService.run_once`
Gate fleet	[code], [deploy]	`refinery_sefer_gate.py`; `ops/sefer-gate/`; reviewed controller document on `main` (2026-10-12)
Host replacement and status	[code], [notes]	`replacement_driver.py`, `replacement_store.py`, migration `0117`; Chapter 11; agents' working notes 2026-11-10 to 2026-11-15
Receipts and journals	[code]	`planning_publication.py`; `refinery_git.target_push_effect_key`; `replacement_effect_journal`; `replacement_provider.py` receipts; trigger on `replacement_quiet_gap_observations`
Rules registry counts	[code]	`rules_contract.py`: `RULES`, `NEVER_CHECKED_RULES`, `Assurance`

Not verified:

- Whether `yesod mail send --as` with the human address is accepted was read from source and not executed; the sentence is marked inferred.
- Whether every agent on the control machine satisfies `_authorized_for_all_stop` through the user-name fallback is inferred, not confirmed against the deployment.
- Which of the ten primed roles had live sessions in November is not established; the working notes speak almost only of the later cast.
- "Parts were rehearsed on disposable machines" rests on the agents' working notes; completion of a full end-to-end rehearsal is not confirmed.
- The statement that integrity failures were few and availability failures many is the editorial reading that Chapter 14 argues; it was not re-derived here.

3 The Two Currencies

Yesod coordinates work with two first-class currencies: **notes** and **beads**. Notes carry intent, lifecycle and dispatch history. Beads carry work structure and dependency readiness. They are linked, and they are not interchangeable.

This chapter explains what each one records, where it is stored, and which questions it can answer. It also corrects a habit that the two-currency picture invites, which is to treat "the state of this work" as one value. In the current factory that question has four separate answers held in four stores, and an operator who reads only one of them will be wrong in predictable ways.

3.1. Notes: Capturing Intent

Every piece of work in Yesod begins as a note. Before a plan exists and before a runner is involved, someone or something captures an intent: a feature to build, a bug to fix, a fact worth remembering. The note is the factory's memory of why a change should happen and, later, of what happened while making it.

A note is a row in the PostgreSQL table `tool_notes`, attached to a row in `tools`. Its identifier, such as `ys-yes-k7q2` (illustrative), has three parts:

- `ys` is the Yesod namespace, shared by every note in the registry.
- `yes` is the project prefix, a short code for the tool the note belongs to. `yes` is Yesod itself; other tools use prefixes such as `sjb` or `als`. The prefix is the tool's stored `tools.note_prefix` when one is set, and a derived three-character prefix otherwise (`db.configured_tool_prefix`, `db.tool_prefix`). No two tools may share a prefix.
- `k7q2` is a four-character suffix drawn from lowercase letters and digits (`_NOTE_ID_ALPHABET` in `db.py`). It is base 36, not hexadecimal, which gives $36^4 = 1{,}679{,}616$ ids per prefix according to the comment in `db.generate_note_id`.

A note has one of exactly three kinds, enforced by a CHECK constraint on `tool_notes.kind`: `feature-request`, `bug-report` and `usage-note`. In the software factory, one feature-request or bug-report note represents one independently mergeable feature. Its important fields include the narrative body, the lifecycle status, the disposition, the linked root bead (`bead_id`), prerequisite notes (`depends_on`), the dispatch target (`agent_dispatch`), acceptance checks, host pins, retry metadata, and a revision counter (`note_revision`) that advances when the reviewed content changes. Body edits are guarded and recorded in `note_edits`.

3.1.1. Status: how far the work has progressed

Feature requests and bug reports move through the **implementation lifecycle**, defined once in `IMPLEMENTATION_LIFECYCLE` in `note_lifecycle.py` (contract version 2). It has twelve states: `captured`, `planning`, `planned`, `open`, `claimed`, `in_progress`, `awaiting_verification`, `awaiting_merge`, `merging`, and the terminal states `done`, `wontfix` and `superseded`. Usage notes use a smaller lifecycle of `current`, `stale` and `superseded`.

Each transition in the contract names who may take it. A traced bug note for the factory's own tool shows the main path:

Step	Status change	Authority
1	born `captured`	creation
2	`captured` → `planning`	`user`, `planning_controller`
3	`planning` → `planned`	`planning_validator`
4	`planned` → `open`	`user`, `auto_promoter`
5	`open` → `claimed`	`runner_claim_transaction`
6	`claimed` → `in_progress`	`runner`
7	`in_progress` → `awaiting_merge`	`runner_finalizer`
8	`awaiting_merge` → `merging` → `done`	`refinery`

At step 3 a validated plan is bound to the note's current revision; a later material edit sends the note back to `captured`. At step 7 the alternative is `awaiting_verification`, used when the outcome is ambiguous or intentionally has no commit. Chapter 5 (Capture, Plan, Deliver) covers these transitions and the recovery edges in detail. Status writes are required to go through `note_status_writer.write_status`, which adds a row to `note_status_changes` and syncs the linked root bead on a best-effort basis. The central writer exists because raw `UPDATE` statements once bypassed both (module docstring).

Step 8 hides the most important change since the August edition. A pushed candidate does not make the note `done` if the factory cannot confirm that the merged code was deployed. In that case the merge attempt ends in the terminal state `merged_not_deployed`, and `refinery_intake._project_merged_not_deployed` writes `tool_notes.status = 'blocked'` with disposition `blocked`. `blocked` is not one of the twelve contract states; `note_status.is_terminal` treats it as a readable, non-terminal value. There is no `awaiting_deploy` status. Chapter 10 (Merged Is Not Deployed) owns this state and its consequences. For this chapter the point is that `blocked` is now a routine place for a merged note to sit, and it is not `done`.

3.1.2. Disposition: whether automation may act

Status is one axis. A note also carries a **disposition**, which answers a different question: should automation be acting on this note at all right now? `disposition.DISPOSITIONS` lists six values: `active`, `held`, `blocked`, `needs_rework`, `quarantined` and `deferred`. Every value except `active` vetoes claiming, gate enrollment and merging (`NON_ACTIVE_DISPOSITIONS`, applied by

`can_claim`, `can_gate` and `can_merge` in the same module). `active` is necessary and never sufficient.

The column is nullable, so a missing value is derived at read time by `disposition.derive_disposition`: `held` if a `hold_reason` is recorded, `blocked` if the legacy status is `blocked`, otherwise `active`. Writes go through `note_status_writer.write_disposition` and are recorded in the append-only table `note_disposition_changes`, whose trigger `trg_note_disposition_changes_immutable` rejects updates and deletes.

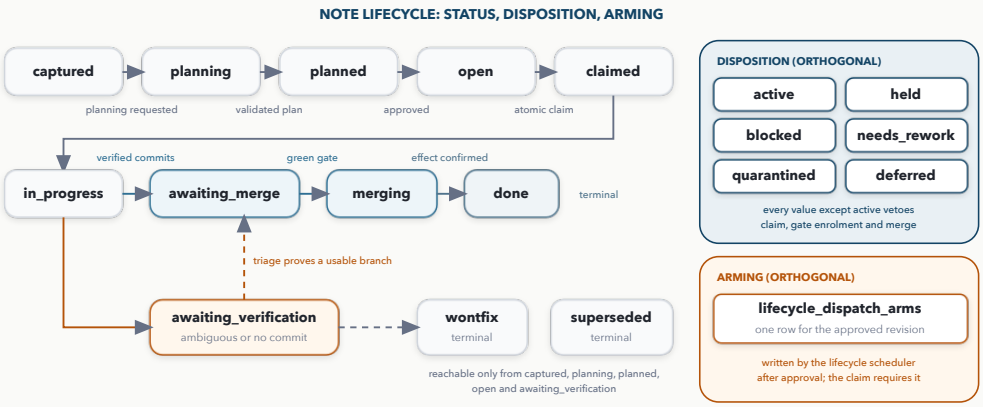

Figure 3.1.: The note lifecycle with its two side axes: disposition, which can veto automation at any status, and arming, which records the dispatch identity of an approved plan.

> **Terminology Trap — One hold, not two.** The August edition described two holds: an `agent_dispatch = 'hold'` sentinel and the `held` disposition. Only the disposition holds a note today. The docstring of `derive_disposition` states that the sentinel is "NO LONGER consulted", that an earlier migration backfilled sentinel rows to `disposition = 'held'`, and that the deprecated writer was removed. The constant `HOLD_DISPATCH_VALUE` remains an allowed `agent_dispatch` value and the claim SQL still excludes it, but an operator holds a note with `note-hold`, which sets the disposition. That no code path still writes the sentinel rests on that docstring and was not traced.

3.1.3. Arming: the third axis

An `open` note is not claimable until it is **armed**: given a dispatch target and a durable record of exactly what was approved. For lifecycle v2 notes, arming is done by `LifecycleScheduler` after approval, not by hand. The note row keeps the current dispatch identity as a projection. The durable record is a row in the table `lifecycle_dispatch_arms`, which binds the note revision, plan fingerprint, refinery lane, repository, target branch, source commit and dispatch target. An armed note is still not automatically claimable; retry delay, dependencies, host routing and runner capacity must also permit the claim. Chapter 5 covers arming and the claim predicate.

3.2. Beads: Planning the Work

If a note says what is wanted and why, a **bead tree** says how the factory intends to get there. Once an intent is captured, a planner breaks it into concrete, ordered steps and the plan is published as a small tree of beads.

Beads is an open-source, dependency-aware issue tracker by Steve Yegge, also used by his Gas Town project, available at github.com/steveyegge/beads. Its command is `bd`. Yesod uses it as the planning substrate beneath every registered tool.

A bead is a task in a per-tool Beads workspace. Every mergeable feature note links to one **root bead**. The descendants of that root are the feature's internal checklist: source changes, tests, documentation, or other concrete steps that should land together. Bead dependency edges order that checklist.

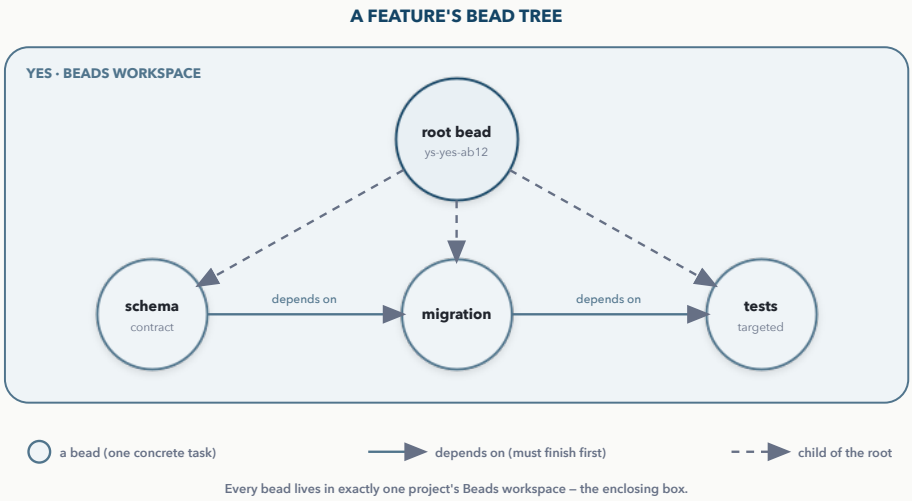

Figure 3.2.: A feature's bead tree: circular nodes are beads, directed edges are dependencies that order the work, and every tree belongs to one project's Beads workspace.

3.2.1. Where beads live

Beads workspaces are stored in Dolt, a versioned SQL database. As of 2026-11-19 the deployment uses one shared Dolt SQL server with one database per tool, named `beads_{tool}`, reached by a least-privileged SQL user that is not meant to create databases; a separate privileged helper provisions each database (`docs/beads-doltsvr-provisioning.md` on `main`). According to the operator's standing instructions to agents, a bead write is durable when the command returns, there is nothing to push, and a `dolt push` from an agent is refused by the server's permissions. That last behavior is not stated in the provisioning document.

3.2.2. Why every call is routed

Agents never run `bd` directly. Every call goes through Yesod:

```
yesod bd <tool> -- show <bead-id>
yesod bd <tool> -- children <bead-id>
yesod bd <tool> -- dep list <bead-id>
yesod bd <tool> -- ready --json
```

The routed path through `beads.run_bd` does four things a raw call would skip. It resolves the tool's workspace and the Dolt connection through one precedence chain (`beads.resolve_dolt_connection`), and a server-mode workspace with an incomplete configuration fails with `DoltConnectionConfigError`, which names the missing keys and never their values. It builds a scoped environment (`_bd_subprocess_env`) that sets the workspace and actor and strips issue-tracker credentials unless the command needs them. It forces a very wide `COLUMNS`, because `bd` wraps text at the terminal width and an inherited width of 80 once inserted raw newlines into JSON output; the comment in the code records that this caused "factory-wide claim starvation". And it logs the command.

Routing also guards the one `bd` operation that touches a Git checkout. `yesod bd <tool> -- worktree create ...` runs `worktree_source.resolve_worktree_source` before anything is created. The selected source checkout must match the tool's expected remote and be clean, and an existing branch of the requested name must already point at the start commit. Any ambiguity is an error with a named code such as `source_ambiguous`, `source_dirty`, `remote_mismatch` or `branch_conflict`, and the command exits before mutating. An operator can pin the source with `--source-root` placed before the `--` separator.

> **Factory Floor — The worktree cut from the wrong clone.** Before 2026-08-15, a routed worktree command invoked from one checkout could attach the new worktree to a different, configured clone at a stale commit. The branch then lacked the code the operator was looking at, with nothing to say so. The fix (commit `5b9e0264c`, tracked as yes-qdve) replaced the guess with the fail-closed preflight above, which reports the exact root, remote and start commit and records an audit row before `bd` runs. The lesson generalizes: when a command must choose among several plausible sources, it should refuse and list the candidates.

3.2.3. Plan-bound leaves

In the August edition, the bead tree was something a worker could extend as it went. For planned work that is no longer true. A child bead published by the planning validator is a **plan-bound leaf**: it carries the identity of the plan that produced it, and the worker assigned to it may not change the shape of the graph. The routed adapter refuses to create descendants under it (`PlanBoundLeafMutationError`) and refuses an unscoped `ready --claim` (`PlanBoundLeafClaimScopeError`). The worker is also instructed not to close the leaf itself (the plan-bound variant of the worker canon, Chapter 4); no adapter check for this was found, and the canon says the runner or refinery closes it after verifying the result. A child

that finishes without a commit reports a structured outcome instead (Chapter 7, Governed Agent Execution).

> **Invariant — A published plan graph is fixed.** The plan a human or policy approved is the plan that runs. `PlanBoundLeafMutationError` in `beads.py` enforces this at the routed boundary, so a worker using the routed command cannot add beads that were never reviewed.

3.3. Two Dependency Layers

Bead dependencies order work inside one feature tree. They answer questions such as whether the query must exist before the CLI wrapper is written. They do not create a new branch or a merge boundary.

Feature ordering belongs to the note layer. A note's `depends_on` field names prerequisite feature notes:

```
yesod tool <tool> note-depends <feature-note> --on <prerequisite-notes>
yesod notes dep list <feature-note>
```

The first command mutates the note; the second is read-only. An API feature and a front-end feature may both depend on a database feature, while that feature's own migration and test beads use ordinary bead dependencies.

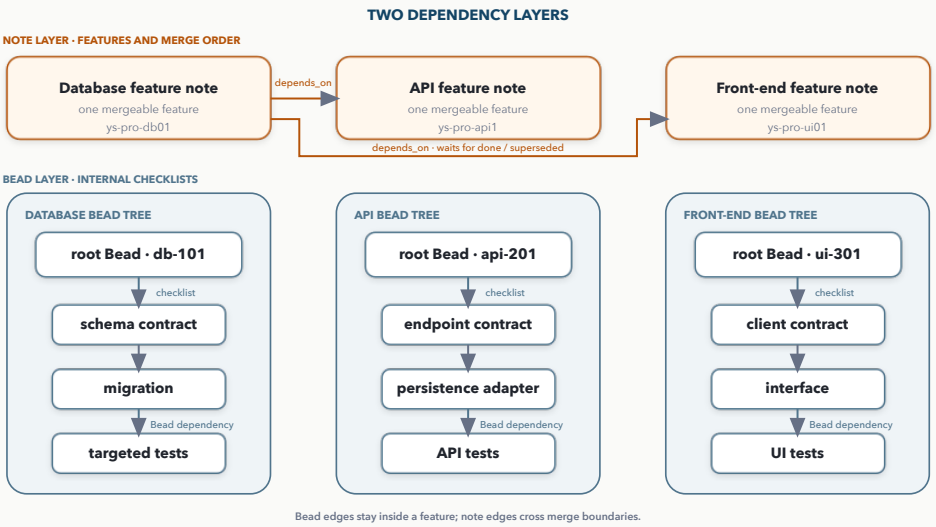

Figure 3.3.: Note dependencies order mergeable features, and bead dependencies order checklist work inside each feature.

The boundary rule is unchanged. If a planned child must start from another change already merged to `main`, must be routed independently, or deserves its own merge decision, it is a

separate feature note with its own root bead, connected by `depends_on`. Ordinary checklist beads do not get child notes.

3.3.1. What clears a dependency

The August edition said a dependency releases when the prerequisite is `done` or deliberately `superseded`. The scheduler is stricter. `lifecycle_dispatch.SUCCESSFUL_PREREQUISITE_STATUSES` contains only `done`. A prerequisite that ends `superseded` or `wontfix` is terminal and unsuccessful, and the dependent note is reported as blocked on a dependency that can no longer succeed.

Two other places in the code use a looser rule. The runner's claim predicate, `_note_dependencies_satisfi` in `claimable.py`, accepts `done` or `superseded`, and `SETTLED_DEPENDENCY_STATUSES` in `note_dependency_policy.py` is the same pair. The help text of `note–depends` also still describes the looser rule. The sets disagree on `main`. Because a lifecycle v2 note must be armed by the scheduler before a runner can claim it, the stricter rule is inferred to win for planned work and the looser one to matter only for legacy notes; this has not been traced end to end.

A dependency edge must also be satisfiable at all. `note_dependency_policy.target_kind_eligibility` refuses a target that can never reach `done`, such as a usage note, and a missing target fails closed. `yesod notes dep integrity` is a read-only check for stored edges that can never be satisfied.

> **Invariant — Dependencies clear on `done` only.** For scheduled work, a prerequisite satisfies a dependent only at status `done` (`SUCCESSFUL_PREREQUISITE_STATUSES`). An upstream agent finishing, a branch awaiting merge, and a merge that has not been deployed all leave the dependent parked.

3.3.2. Merged but not deployed parks the dependents

The last case in that invariant has a cost. A prerequisite whose merge attempt ended `merged_not_deployed` sits at `blocked`, so a follow-up note that depends on it cannot be armed even though the code it needs is on `main`. The agents' working notes for 2026-10-24 record this as the normal condition at that time, and record the workaround in use: clear `depends_on` before planning, and make the first acceptance check of the first commit-bearing child a search for something the prerequisite introduced, so the child stops with a `blocked_on_dependency` outcome if the code is absent. A read-only measurement of what a merge-aware rule would decide exists in `note_settlement_read.py`, whose docstring says it "deliberately does not participate in any dispatch decision". Chapter 10 tells this story in full.

3.4. Dispatch Lanes, Hosts and Affinity

A **dispatch lane** is a named execution target (a model and the command used to invoke it), not a machine. Each runner advertises the targets it can serve, and a note's `agent_dispatch` names one. (The word "lane" has two other meanings in the factory; Chapter 16, Cost-Aware Dispatch, separates them.)

The claim predicate also respects routing constraints. A hard host pin (`pin_host` with `pin_priority = 'hard'`) keeps a task on the one machine that has what it needs, and a soft pin prefers a host and yields after `DEFAULT_DISPATCH_SOFT_AFFINITY_SECONDS = 300`. A failed task waits for `next_retry_at`. Chapters 5 and 6 cover these in context.

3.5. Four State Machines, Not One

When an operator asks what state a piece of work is in, the answer may need four lookups:

1. **The note**: lifecycle status plus disposition, with the arm row for dispatch identity.
2. **The bead graph**: open or closed, ready or blocked by dependencies, and for closed plan-bound leaves, the close reason.
3. **The merge attempt**: the refinery's integration state machine, one immutable record per try, with active states such as `gating` and `pushing` and terminal states such as `succeeded`, `gate_red` and `merged_not_deployed`. Chapter 8 (The Merge Controller) lists all of them.
4. **The replacement intent**: whether the merged code is running, which for the controller means a host swap tracked by its own durable state. Chapter 11 (Deploy Is a Host Swap) describes it, including the fact that no such swap has completed automatically in production as of 2026-11-19.

These views can differ without the system being incoherent. A note can remain `awaiting_merge` while its latest attempt is `gate_red`. A note can be `blocked` while its attempt is terminal and its commit is on `main`. A bead can be closed while no test ever ran for it. Reconciliation does not force every store into one vocabulary; it repairs the relationships that should hold between them.

> **Operator Check — Ask the right store.** Use note status and disposition for lifecycle (`yesod tool <tool> note-why <note>` is read-only), Beads for readiness and close reasons, the run ledger for execution, merge attempts for integration, Git ancestry for what landed, and deployment evidence for what is running. Do not use `note-plan` to look at a plan: it requests a new one.

Notes and beads explain how the factory coordinates code work. They sit inside a broader registry that also preserves tool knowledge and reusable processes, which the next chapter describes.

Source Notes

Claim or section	Tag	Anchor
Note ids, prefixes, kinds	[code]	`db.py`: `generate_note_id`, `_NOTE_ID_ALPHABET`, `tool_prefix`
Lifecycle states, authorities	[code]	`note_lifecycle.py`: `IMPLEMENTATION_LIFECYCLE`, `CONTRACT`
Central status writer	[code]	`note_status_writer.py` docstring
`blocked` after undeployed merge	[code]	`refinery_intake._project_merged_not_deployed`
Six dispositions, derivation, audit	[code]	`disposition.py`; migration 0120
Hold sentinel retired	[code]	`derive_disposition` docstring; `HOLD_DISPATCH_VALUE`
Arming	[code]	`lifecycle_dispatch.py`; `lifecycle_dispatch_arms` (migration 0050)
Dolt topology	[deploy]	`docs/beads-doltsvr-provisioning.md`, as of 2026-11-19
No push needed; push refused	[notes]	operator's standing agent instructions
Routed `bd`, `COLUMNS`	[code]	`beads.py`: `run_bd`, `_bd_subprocess_env`
Worktree preflight, incident	[code]	`worktree_source.py`; commit `5b9e0264c`
Plan-bound leaves	[code]	`beads.py`; commits `2a00332e5`, `8ab2420fe`
Dependency rule and mismatch	[code]	`lifecycle_dispatch.py`; `claimable.py`; `note_dependency_policy.py`
Which rule wins for v2 notes	[inferred]	not traced end to end
Undeployed prerequisite workaround	[notes]	agents' working notes, 2026-10-24
Settlement read is shadow only	[code]	`note_settlement_read.py` docstring
Pins, affinity, host plans	[code]	`claimable.py`; `host_plans.py`

Not verified:

- Whether any code path still writes `agent_dispatch = 'hold'`; the `derive_disposition` docstring says the writer was removed.
- Whether the routed adapter refuses a worker's `close` of a plan-bound leaf; only the creation and claim-scope refusals were found in `beads.py`.
- That a `dolt push` is refused by server permissions; the provisioning document describes the least-privileged user only.
- Whether a note with a `superseded` prerequisite can be armed and claimed on any path; the scheduler and the claim SQL disagree on `main`.
- Whether `merged_not_deployed` is still the outcome of most production merges as of 2026-11-19; the evidence is the working notes of 2026-10-24.

- The date of the terminal-width claim starvation and of the raw-status-write incident; both come from code comments without dates.
- The live lifecycle rollout stage (no database was read).

4 The Registry Beneath the Factory

The software factory is built on an older and more general idea: a personal registry in which tools can be described, questioned, composed, observed and improved together. That layer is easy to lose sight of once runners and gates enter the story, and it is what makes Yesod more than a job queue with a model attached.

This chapter describes the registry: tools and their notes, the two question-answering commands and how they ground their answers, processes, topics, and the two pieces that connect the registry to the factory. Those are repository onboarding, which decides whether a repository may be worked by agents at all, and the worker canon, the single source of the instructions every worker agent receives.

4.1. Tools as Durable Context

Picture an agent in the middle of a job. It is assembling a blog post and calls `scr-wrap`, a screenshot-wrapping tool, to turn a raw screen capture into a framed image. On this run `scr-wrap` fails.

A naive agent gives up here, or papers over the failure and ships a broken post. This agent knows that `scr-wrap` is a registered tool, so the failure is something it can act on. It files a bug report against the tool (a mutating command):

```
yesod tool scr-wrap bug-report \
  "wrap fails on PNGs wider than 2048px: 'invalid dimensions'"
```

Yesod records the report and returns a note id, say `ys-scr-4f2a`. The id is a ticket: it is addressable and it has a status. The agent does not have to hold the problem in its context or block. It keeps the id, sets the screenshot subtask aside, and continues with the parts of the post that do not depend on it. If the tool's repository is managed by the factory, the report can flow through the machinery the rest of this book describes: a plan, a bead tree, a runner, a gate, a merge. The waiting agent can check on it with a read-only command:

```
yesod notes show ys-scr-4f2a --json
```

Two cautions keep this picture honest. Yesod does not estimate when a fix will land; the August edition suggested it would, and nothing on `main` does. And a status of `done` records

that the work was completed and settled; it does not prove that the fixed tool is the one installed where the agent is running. The system prompt for `yesod ask` says so in its own words: a note whose lifecycle is "done" records completion and "is not a deployment receipt". The agent's last step is therefore to run `scr-wrap` again and see.

Even with those limits, no human relayed the bug between two tools, and the problem did not die in a transcript.

A registered tool is a row in `tools` with a name, a kind, a description, a repository or URL, binary registrations, and a stream of timestamped notes. There are four kinds: `internal` (tools Stephen owns), `team` (built by colleagues), `external` (public or third-party), and `topic`, which is how cross-cutting knowledge is stored and is covered below. The basic operations are ordinary:

```
yesod tools add NAME --kind internal --description "..."
yesod tools list
yesod tool NAME usage-note "A durable operating fact"
yesod tool NAME feature-request "A capability this tool should gain"
yesod tool NAME bug-report "A behavior that needs repair"
yesod tool NAME notes --detail
```

All but `tools list` and `notes` write to the registry. A tool is not an active agent and does not file complaints by itself. Agents and workflows acting in one tool's context can file notes against any other registered tool, and that is what makes integration friction durable. Tools learn about each other in Yesod by accumulating addressable, attributable facts in a common registry, not by sharing mutable prompts.

4.2. Asking the Registry

Two commands answer questions from the registry, and they assemble their evidence differently.

`yesod query TOOL -q "..."` is about one tool in its ecosystem. Its context loader, `query._load_context`, gathers the tool's metadata, all of its notes, cached snapshots of its beads (`beads_issue_snapshots`), and every registered process that references the tool, including that process's other steps. The process context matters because a tool rarely has one meaning in isolation.

`yesod ask` works across the whole catalog. It answers questions such as which existing tool fits a job, what the preferred way to publish an episode is, or which tool has an unresolved bug that makes a workflow risky. The August edition described it as packing the entire catalog into the prompt. That is now the `--full` option. By default the context is built by retrieval (`retrieve_catalog` and `pack_catalog` in `retrieval.py`):

- Compact indexes of every tool and every process are always included, so the model knows what exists.
- Tools, notes and processes are ranked against the question, and the top-ranked ones get their full bodies, within a token budget (`DEFAULT_TOKEN_BUDGET = 5000`).

- Ranking is a hybrid. A PostgreSQL full-text rank over the question's words and an embedding cosine similarity are fused by weighted reciprocal rank fusion, with the semantic rank weighted 2.0 and the lexical rank 1.0 (`ASK_RRF_SEMANTIC_WEIGHT`, `ASK_RRF_LEXICAL_WEIGHT`, `ASK_RRF_RANK_CONSTANT = 60`).
- Usage notes are preferred as factual authority unless the question is about work items (`retrieval.prefer_usage_note_authority`).
- Superseded notes are excluded unless the caller passes `--history`, and then they are shown with links to their successors.

The embedding search the August edition placed on the roadmap is therefore in code. Vectors are stored in `content_embeddings` and precomputed by `yesod render-embeddings`. Which embedding provider and model production uses was not verified.

The answer contract is the part agents rely on. The system prompt requires a `## Sources` section listing only what was actually cited: notes by id, kind and date, and comments and status changes by anchors of the form `<note-id>#comment-<row-id>` and `<note-id>#status-<row-id>`. With `--json` the model must return four fields: `answered`, `confidence` (`high`, `medium` or `low`), `answer` and `sources`. When the catalog does not cover the question, the required reply is `answered` false with empty sources. Worker agents are instructed to run `yesod ask` before guessing about an internal tool or flag and to trust either the cited answer or the honest refusal. The prompt also keeps a soft preference among tools that would all work: internal first, then team, then external, unless a better fit is clearly external.

> **Design Trade-off — Retrieval over omniscience.** Yesod does not ask a model to remember a private tool ecosystem. It selects current notes and process relationships at query time and requires citations, so the model reasons over evidence the user can inspect. The price is that an answer is only as good as the ranking, which is why whole-catalog packing was kept as `--full`.

> **Factory Floor — The vectors that never died.** Until 2026-08-20, `content_embeddings` had no delete path. Vectors for superseded and removed notes stayed in the table (the commit message says the live catalog had accumulated thousands of dead rows), and ranking fetched the 4,096-dimension vectors and scored them in Python. Commits `eb172aa80` and `058fb8501` added optional in-database vector storage and deleted vectors at the commands that remove content (`supersede`, `note-rm` and their relatives, in `embedding_hygiene.py`). A retrieval layer needs a deletion path as much as an indexing one.

4.3. Processes as Molecules

A Yesod **process** is a reusable arrangement of tool actions. Early processes were ordered lists. Current processes can also be directed acyclic graphs.

```
yesod processes add release-booklet \
  --description "Render, inspect, and publish a booklet" \
  --step 'pandoc:render:build the PDF' \
  --step 'chunkpdf:inspect:check page boundaries' \
  --step 'shup:publish:upload the approved artifact'
```

Without explicit dependencies, the steps form a sequential chain. A declaration such as `--dep 3:1,2` says that step 3 depends on steps 1 and 2 and switches the process to graph authoring. Independent steps become parallel branches. A step can also be a human handoff.

The word **molecule** comes from Beads. Under Architecture Decision Record 005, the source of truth for a process is a reusable Beads prototype in the reserved `proc` workspace (`PROC_WORKSPACE_NAME` in `beads.py`). The PostgreSQL tables `processes` and `process_steps` mirror that graph for catalog queries and the UI. A process therefore has two forms: a readable catalog description for search and composition, and an executable dependency structure that can be instantiated, or **poured**, into a concrete run. Process runs get their own note ids under the reserved `proc` segment, such as `ys-proc-<4 chars>` with dotted step suffixes.

The command vocabulary is small:

- `yesod processes validate` checks tool references and structure.
- `yesod processes pour PROCESS` instantiates a process with `bd mol pour` and creates run notes.
- `yesod processes runs` lists concrete pours.
- `yesod processes squash RUN` compacts a completed molecule into a durable digest.
- `yesod schedules` registers recurring pours.

This part of the system was stable through the autumn: no commit since 2026-08-20 changed the process modules other than cross-cutting ones.

4.4. Composition Creates a Feedback Network

Once tools and processes share a registry, several feedback paths open. Querying a tool reveals the processes that rely on it. Usage notes record the command that works, a rate limit, or a required ordering. `yesod call TOOL ...` runs a command and stores the invocation, with output, exit code and a session identifier, in `tool_calls`. An agent running step four of a process can file a bug against the tool from step two.

4.5. Topics, and the Bulletin Over Them

Not every durable fact belongs to one tool. A **topic** holds cross-cutting knowledge such as factory authority, gate operations or documentation standards. In storage, a topic is a `tools` row with `kind = 'topic'`. It has a `topic_kind` of `episodic` (a bounded investigation) or `standing` (evergreen knowledge), a `topic_status` of `open`, `active` or `concluded`, and a list of related tools (`topic.py`).

The ownership rule is simple:

- put tool-specific behavior on the tool;
- put reusable fleet-wide practice on an existing canonical topic;
- put concrete multi-step execution in a process;
- put implementation work in a feature request or bug report linked to Beads.

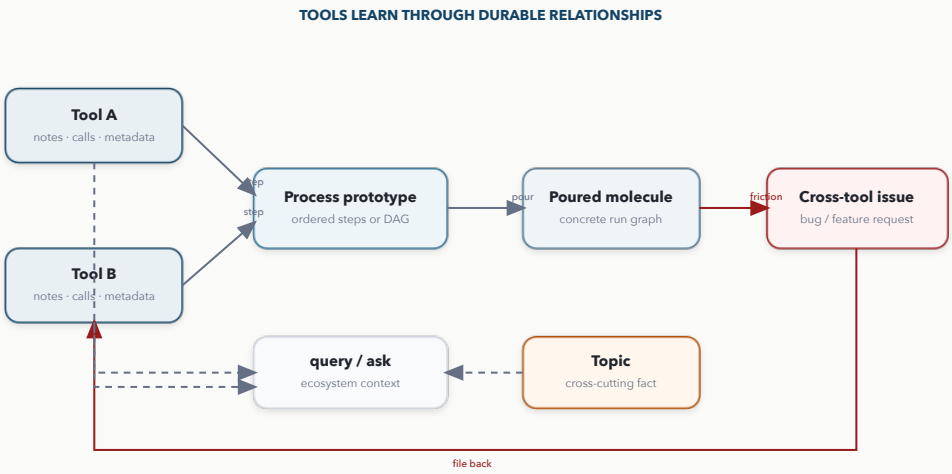

Figure 4.1.: Tools, processes, concrete runs and cross-tool feedback form a learning loop.

The rule keeps the registry from degenerating into one notes table whose entries cannot be retrieved correctly.

Since September, topics also carry the factory's main coordination channel. The **bulletin board** is, in the words of its own usage text, "an opinionated read/write surface over topic notes and note comments". A bulletin post is a topic note with a tag marking it as a bulletin (`BULLETIN_TAG`), a kind, and optional typed links to notes, beads, gate runs and mail. Topics own board membership; the default board is `DEFAULT_BOARD_NAME = "b/yesod"` (`bulletin_board.py`). Post bodies, titles and replies reuse the catalog tables, and separate bulletin tables hold only metadata such as attribution, links and promotion provenance (`bulletin_store.py` docstring). What is posted to the board is therefore catalog state, and `yesod bulletin promote` (a mutating command) turns a post into ordinary catalog knowledge with its provenance recorded. Chapter 17 (The Agent Organization) describes how the agents use the board and how authors are verified.

4.6. Onboarding a Repository

A tool in the registry is knowledge. For agents to change its code, its repository must be **onboarded**: connected to a Beads workspace, a runner that can check it out, and a refinery lane that owns merges to its branch. The August edition did not cover this step, and since then it has become policy-governed.

`yesod onboard --github URL` is one idempotent convergence command (`onboarding_cli.onboard_comma`
`yesod onboard REPO status` is its read-only counterpart, and `yesod onboard status --all` audits the whole fleet. The command checks eleven invariants in a fixed order (`ONBOARDING_INVARIANT_ORDER` in `onboarding_model.py`), from `catalog_identity` and `note_namespace` through `beads_workspace`, `dispatch_runner_coverage` and `refinery_lane`

to `terminal_acceptance`. Each check reports one of seven states, including `ok`, `missing`, `drifted`, `pending` and `unknown`.

Two readiness levels come out of this. A repository is `dispatch_ready` when the catalog, Beads, runner, lane and controller ownership agree. It is `verified` only when, in addition, its lane has completed a real terminal green acceptance attempt. A missing proof is reported as pending and is never synthesized from configuration.

The policy lives in a reviewed file, `config/onboarding-fleet.yaml`. Every catalog entry that has a repository must appear there as `managed` or `excluded`, and an exclusion needs a written reason. On `main` at the edition snapshot the file lists 35 projects, 27 managed and 8 excluded. A repository absent from the file makes the fleet audit red, and `yesod onboard` refuses it and prints a proposed entry for a human to review (`onboarding_fleet.py`). The file's header states the rule: a new repository "is fail-closed until this policy is reviewed and updated".

> **Invariant — No unreviewed repository enters the factory.** Admission to the fleet is a reviewed line in `config/onboarding-fleet.yaml`, enforced by the admission check in `onboarding_fleet.py`. Running the onboarding command is not enough.

Activating a lane does not wait for a release, but it is bound to a reviewed commit: the lane file installed on the controller must equal the committed `config/refinery-lanes.json` byte for byte (`onboarding_activation.py`). The forge layer is neutral (`forge.py`), with GitHub first class and GitLab added as a second backend in September and October.

> **Proven Where — as of 2026-11-19.** The onboarding command: the plan document on `main`, dated 2026-08-09, carries the status line "deployed and verified in production". The reviewed fleet policy and the GitLab backend: on `main`; how many of the 27 managed repositories report `verified` was not read. A hosted repository-import service sharing these invariants: on `main`, dormant, not known to run anywhere (Chapter 21, The Road Ahead).

4.7. The Worker Canon

The registry also holds the instructions that tell a worker agent how to use it. They used to be maintained in four places that drifted independently, according to the docstring of `worker_canon.py`: the runner's prompt builder, the worker primer, the repository's `AGENTS.md`, and the bundled skill file. That module, introduced in June (commit `27388daf0`), is now the single source, and four consumers render from it:

- the runner's dispatched prompt, with the real note id, tool and bead id substituted;
- `yesod prime --worker`, with the placeholders left literal;
- a generated block in `AGENTS.md`, refreshed by `yesod render-agents-md` from a pre-commit hook;
- the bundled skill file, filled at install time.

The canon has eleven sections (`SENTINELS`): a command crib for `yesod` and `yesod bd`, the workflow, tooling standards, writing tests that pass the gate, test-database hygiene, scratch VM policy, the progress heartbeat, git discipline, worktree conventions, root bead ownership, and the write-habit rule. The last one is the ownership rule from the topics section, issued

as a standing instruction: when a worker learns a durable fact about a tool, it files the fact back to the tool or to an existing topic. A second variant, selected for plan-bound leaves, replaces three of the sections (the command crib, the workflow and root bead ownership) with versions telling the worker not to create descendant beads and not to close its leaf. Each section begins with a fixed heading, and the module's docstring says a test asserts that every consumer's output contains every heading, so a consumer that drops a section fails in CI.

Chapter 7 (Governed Agent Execution) covers what the rules say and why. Instructions to agents are part of the system's configuration, and four hand-edited copies of them meant agents running under four slightly different rule sets.

4.8. More Than a Factory Front End

The registry remains useful when no coding agent is running. The software factory is one consumer of it, the one that turns selected notes into code.

> **Invariant — The catalog is not merely a queue.** Work may begin as knowledge, remain knowledge, become a process, or become dispatchable implementation. This is held by design and convention (the three note kinds and the ownership rule), not by one enforcing function.

Together these layers explain how a request can move through the factory without living in any one agent's context. The registry preserves intent and tool knowledge, Beads supplies work structure, deterministic services enforce progression, and Git proves what landed. Part II returns to the request from Chapter 1 and follows each boundary in detail.

Source Notes

Claim or section	Tag	Anchor
Tool and note commands, four kinds	[code]	`tools.py`, `tool.py`, `notes.py`; `db.py` DDL
"done" is not a deployment receipt	[code]	`ask._SYSTEM_PROMPT_TEMPLATE`
`yesod query` context	[code]	`query._load_context`
`yesod ask` retrieval and ranking	[code]	`retrieval.py`: `retrieve_catalog`, `ASK_RRF_*`
Citations, JSON contract	[code]	`ask.py`: system prompt, `_JSON_SYSTEM_SUFFIX`
Embeddings and hygiene incident	[code]	`embeddings.py`; `embedding_hygiene.py`; commits `eb172aa80`, `058fb8501` (2026-08-20)
Processes, `proc` workspace	[code]	`processes.py`; `beads.py`; `db.py`
`yesod call`, `yesod schedules`	[code]	`call.py`; `schedules.py`
Topics	[code]	`topic.py`: `TOPIC_KINDS`

Claim or section	Tag	Anchor
Bulletin over topic notes	[code]	`bulletin_board.py`; `bulletin_store.py`; `prime.BULLETIN_USAGE_ADDENDUM`
Onboarding command and invariants	[code]	`onboarding_cli.py`; `onboarding_model.py`
Fleet policy and counts	[code]	`config/onboarding-fleet.yaml`; `onboarding_fleet.py`
Onboarding deployed	[deploy]	plan doc of 2026-08-09 on `main`
Lane activation, forge layer	[code]	`onboarding_activation.py`; `forge.py`
Worker canon	[code]	`worker_canon.py` docstring, `SENTINELS`

Not verified:

- Which embedding provider and model production uses, and whether in-database vector ranking is active there.
- How many managed repositories report `verified` as of 2026-11-19.
- Whether the hosted repository-import service runs anywhere; code-intel describes it as dormant by default.
- That `tests/test_worker_canon.py` passes; the claim rests on the module docstring, and no test was run.
- Current counts of registered tools, notes and topics (no database or CLI was read).
- The `scr-wrap` scene is an illustration; the tool name and note id are not taken from the live catalog.

Part II.

From Intent to Merge

5 Capture, Plan, Deliver

This chapter follows one request from the moment it is written down to the moment a runner claims it. It names every state the note passes through, the table that records each step, the verb that causes it, and who or what is allowed to cause it. The mechanism is called Lifecycle v2. Its state names have not changed since August 2026. Almost everything added since then is fencing, recovery and diagnosis, and the reasons for those additions are the useful part of the chapter.

Three points matter most. A plan is bound to an exact revision of the note and to a fingerprint of everything the planner read. Approval and arming are separate steps performed by different actors. And some of what stops work here is code, while some is operator policy. Two practices that sound like mechanisms, "hold every delivery" and "pause the factory", are narrower than their names suggest, and the chapter says exactly how narrow.

5.1. Three Fields, One Note

A feature-request or bug-report note in the `tool_notes` table carries three independent fields. Chapter 3 (The Two Currencies) defines them in full; this chapter needs only their roles.

The **status** says where the work is. `IMPLEMENTATION_LIFECYCLE` in `note_lifecycle.py` defines twelve values: `captured`, `planning`, `planned`, `open`, `claimed`, `in_progress`, `awaiting_verification`, `awaiting_merge`, `merging`, and the terminal `done`, `wontfix` and `superseded`. `CONTRACT.version` is 2, which is where the name Lifecycle v2 comes from.

The **disposition** says whether automation may act on the note at all: `active`, `held`, `blocked`, `needs_rework`, `quarantined` or `deferred`. Every value except `active` vetoes claims and merges (`disposition_eligible_sql` in `disposition.py`) and new planning leases (`note_allows_new_planning_lease` in `planning_jobs.py`).

The **automation policy** says how far the note may travel without a person: `capture`, `plan` or `deliver` (`AUTOMATION_POLICIES` in `note_lifecycle_ops.py`). `POLICY_STOP_STATES` gives the stopping points: `captured`, `planned`, and for `deliver`, "done or a state requiring human authority".

A note is created with one of the three policies. `policy_from_creation_flags` maps the creation flag `--auto-plan` to `deliver`, `--plan` to `plan`, and neither flag to `capture`. `creation_values` starts a `plan` or `deliver` note directly in `planning`; a `capture` note starts in `captured` and waits.

5.2. The Path and Its Authorities

Each edge in `IMPLEMENTATION_LIFECYCLE.transitions` is a `LifecycleTransition` that names the authorities allowed to take it and the cause that justifies it. The first half of the path, which is this chapter's subject, looks like this.

Edge	Named authorities	Cause
`captured` → `planning`	`user`, `planning_controller`	planning requested for the current revision
`planning` → `planned`	`planning_validator`	validated plan bound to the current revision
`planned` → `open`	`user`, `auto_promoter`	current validated plan approved for delivery
`open` → `claimed`	`runner_claim_transaction`	all dispatch predicates pass; atomic claim

There are recovery edges as well. `planning` → `captured` cancels or abandons planning. `planned` → `planning` and `open` → `planning` are explicit re-plans. `planned` → `captured` and `open` → `captured` happen when a material revision invalidates the plan. No edge leads directly from `claimed` or `in_progress` to a terminal state. Started work reaches `done` through `awaiting_merge` or `awaiting_verification`, and can be closed `wontfix` or `superseded` only after it has come back to `open` or `awaiting_verification`.

The authority names are metadata in the descriptor. `LifecycleDescriptor.__post_init__` refuses a transition with no authority or a duplicate edge at import time, and `_require_transition` in `note_lifecycle_ops.py` refuses an edge that does not exist. A search of the Python package on `main`, subpackages included, found no code that compares a caller's identity with the `authorities` tuple; the only reader is the descriptor's own serialization. `note_status_writer.write_status` takes an `actor` string and records it in `note_status_changes`. The real enforcement is elsewhere: each step has its own function with its own preconditions, and those preconditions are what the rest of this chapter describes.

5.3. One Note, Traced

The trace below is constructed from the code path on `main`, not copied from a recorded note. The note id is illustrative. The request is the one from Chapter 1 (Follow the Change).

1. An operator files the request with no planning flag. The note is born `captured`, disposition `active`, policy `capture`, `note_revision = 1`.
2. The operator runs `yesod tool yesod note-plan ys-yes-ab12`. `request_planning` sets the policy to `plan`, writes status `planning`, and `planning_jobs.recover_planning_request` creates a row in `planning_jobs` for revision 1, job generation 1, state `queued`.

3. A runner with `planner_enabled` reaches `_tick_planning` in its poll loop. `acquire_lease` moves the job to `running`. A Codex subprocess (`LocalCodexPlanningProvider`) reads a snapshot of the repository and returns a plan.
4. The plan lists two open questions. Validation fails. The job ends `failed` with terminal reason `invalid_provider_output`. The note stays in `planning` and its disposition becomes `needs_rework`, with a hold reason that starts `planning_invalid_output:`.
5. The operator appends the answers to the note body. `invalidate_for_material_edit` advances the note to revision 2, marks the old job `superseded` with reason `note_revision_advanced`, clears the plan fields, and returns the status to `captured`.
6. The operator runs `yesod codefactory-dispatch replan ys-yes-ab12`. The disposition returns to `active`, the status to `planning`, and a new job is bound to revision 2.
7. This plan validates. A row in `planning_publish_intents` is prepared, `BeadPlanPublisher` writes the child Beads under the note's existing root Bead, and the intent is finalized. The job ends `succeeded`. The note becomes `planned`, with `planned_revision = 2`, a `plan_fingerprint` and a `validator_provenance`.
8. The operator reads the plan and runs `yesod tool yesod note-approve ys-yes-ab12`. `approve` sets the policy to `deliver` and the status to `open`.
9. On its next pass `LifecycleScheduler` re-evaluates the note, inserts a row in `lifecycle_dispatch_arms`, writes the dispatch target into `agent_dispatch`, and issues `pg_notify` on the channel `yesod_note_armed`.
10. The note's row in `claimable_candidates` reads `verdict_authoritative` and `ready`, with the first child as the current child.
11. A runner that advertises the target claims the note: `open` → `claimed` → `in_progress`. Chapter 6 (The Runner) takes over from there.

Steps 2, 5, 6 and 8 mutate state. The sections that follow explain each mechanism in the order the trace meets it.

5.4. Planning Is a Durable Job

Planning is recorded as a job. If the planner process dies, the job row survives and another attempt can be leased.

Identity. A job is keyed by note id, note revision and job generation (`ensure_job`). The revision changes when the note's reviewed content changes. The job generation counts attempts at a plan within one revision: when a job is terminal and no publication is in flight, the next request opens generation plus one. `ensure_job` refuses to reuse a job whose input changed without a revision bump ("planning input changed without a note revision increment").

States. `ACTIVE_STATES` are `queued`, `running`, `retrying` and `blocked`. `TERMINAL_STATES` are `succeeded`, `failed`, `cancelled` and `superseded`. Only `queued` and `retrying` are claimable (`CLAIMABLE_STATES`); `acquire_lease` also recovers a `running` job whose lease has expired.

Input snapshot. The job stores the normalized input it was asked to plan and a fingerprint of it. The current input schema is 3 (`CURRENT_INPUT_SCHEMA_VERSION` in `planning_input.py`). It covers the note body, the acceptance check, the task kind, the target branch, the declared dependencies, the routing fields (dispatch target and complexity) and the note's comments. Host pins are excluded.

Lease. `acquire_lease` locks the note row, takes a fleet-wide advisory lock, and checks the planner concurrency limit (`planner_fleet.read_limit`; `DEFAULT_MAX_CONCURRENT` is 1). It refuses when the note's disposition is not `active` (`note_allows_new_planning_lease`). `PlanningExecutionConfig` sets the defaults: a lease of 300 seconds, renewed by `renew_lease` while the planner works, and an overall deadline of 1,800 seconds.

Attempts. `fail_job` retries with a backoff that doubles from `retry_backoff_seconds` (30) until the attempt count reaches the job's maximum (3 when unset), then ends the job `failed` with `retry_exhausted`. An exhausted job leaves the note `blocked`; invalid output leaves it `needs_rework` (written by `planning_execution.py`, read back by `matching_planning_failure_disposition`).

Terminal reasons. The reason code on a terminal job is the most useful field in this subsystem. The ones an operator meets most often are `validator_accepted` (success), `invalid_provider_output`, `note_revision_advanced`, `stale_note_revision`, `retry_exhausted`, `completed_prefix_invalid`, `planning_root_bead_missing` and `schema_mismatch`.

A held note's job is never leased, and for a long time that was silent. `list_due_jobs` filters through `classify_planning_eligibility`, which reports `planning_disposition_blocks_lease`. The classifier was centralized on 2026-10-13 (`105a1b990`), and read-only inspection arrived on 2026-11-08 (`ee366b9ef`) as `note-plan-status` and `yesod planning-queue`.

5.5. The Plan Contract

The planner is told that it is planning only: no edits, no Beads, no commits (`build_planner_prompt` in `planning_contract.py`). What it returns must satisfy a strict model, `PlanArtifact` in `contracts/planned_root.py`, at `PLAN_CONTRACT_VERSION` "5.0".

A plan names the note (id, revision, input fingerprint), one root Bead, and a list of child tasks. Each `ChildTask` has a key, a Bead id of the form `<root>.<n>`, a title and description, at least one file, at least one acceptance check written as an exact command, dependencies on earlier children only, and a `commit_expected` flag. The validators require at least `MIN_CHILD_TASKS` (3) children, contiguous Bead ids, and at least one child that expects a commit. There is no maximum.

The plan also carries `decisions` and `open_questions`, and these two lists are where most planning failures come from.

> **Invariant — A plan with open questions cannot publish.** `PlanArtifact` rejects any artifact whose `open_questions` list is non-empty, with the message "plan artifact has unresolved open_questions and cannot be published". The job ends `failed` with `invalid_provider_output` and the note goes to `needs_rework`. Nothing re-enters the note automatically.

The rule is sound: a worker should not build on a plan whose author did not know what was wanted. Its first form had a cost.

Factory Floor — The open-questions loop, September 2026. On 2026-09-08 the agents' working notes record 35 notes in `planning` or `planned`, of which 5 were `active`; 14 were `needs_rework`, all with the open-questions reason. On 2026-09-09 the notes count 38 of 39 planning failures in six hours with that cause. On 2026-09-10 they count 48 jobs failed on open questions against 36 that succeeded in 24 hours. Over seven days they count 81 complete plans discarded, and of 77 questions examined, 64 were real design questions and 13 were mechanical ("declare a dependency on this note"). The same notes record that adding two runner seats changed nothing, because nothing in the queue was claimable to plan, and that planner activity went from zero to five within twenty minutes of answering the questions on six notes. All of these figures come from the working notes, not from a measurement kept on `main`. The cause was a gap between two correct rules: a plan with a question may not publish, and a failed plan parks its note. Nothing routed the question to anyone, and nothing returned an answered note to the queue. The code responses came in pieces. A `replan` verb (`3fe504728`, 2026-09-21) became the single supported exit from `needs_rework`. A typed decision contract (`8bf282c70`, 2026-10-03) split planner choices into those it must decide and record (`DECIDE_AND_RECORD_DECISION_TYPES`: naming, wire shape, operational default, test policy, file placement) and those it must ask about (`MUST_ASK_DECISION_TYPES`: product behavior, a security or authority boundary, cost or external quota, a cross-note dependency). A must-ask decision is accepted only with a verbatim citation of the note body or a comment. A fail-closed check needs a path back in, and a queue that stops moving should be read by disposition before anyone adds capacity.

5.6. Bound to a Revision and a Fingerprint

A plan is valid for one exact input. Two different events can invalidate it, and they behave differently.

A **material edit** to the note, such as a body change, calls `invalidate_for_material_edit`. It advances `note_revision`, supersedes older jobs with `note_revision_advanced`, clears `planned_revision`, `plan_fingerprint`, `validator_provenance` and both idempotency keys, and returns the note to `captured`. It is refused while the note is in a live execution state (`LIVE_EXECUTION_STATES`) or while a publication is in flight.

A **comment** does not advance the revision. It does change the input fingerprint, because comments have been planning input since 2026-08-24 (`ceae82033`). If the fingerprint moves while a job is running, publication ends that job `superseded` with `stale_note_revision` (`planning_publication.py`). If it moves after the note is `planned`, the approval check `planning_input_drift` fails. The dispatch target and complexity are in the fingerprint as well, so changing a note's target between `planned` and approval has the same effect. The practice that follows, recorded in the agents' working notes across several October dates, is short: put the full specification in the body before requesting a plan, set the target first if one is wanted, touch nothing until the note reads `planned`, then approve.

Each re-plan derives its children from the body and comments alone. On 2026-11-08, according to the working notes, a requirement that existed only in the plan for a note's revision 4

was absent from the plan for revision 6, and a reviewer caught the omission. A reviewer's finding about a plan has to be written into the note, or it must be found again on the next plan.

Terminology Trap — `note-plan` is a request. `yesod tool <tool> note-plan <id>` calls `request_planning`. On a note that is already `planned` or `open` it clears the plan fields and sends the note back to `planning`. The working notes record a finished plan destroyed this way on 2026-10-13 by an agent that meant only to read it. To read, use `note-why --json` or `note-plan-status`, which mutate nothing.

Factory Floor — Planning that says planning but is dead, 2026-09-11. Two notes that other work depended on had shown status `planning` for hours, with no open questions. Neither had a live job. The working notes for that date give two causes. On one note, comments written to add detail had changed the input fingerprint under a running job, twice, and each time the job ended `stale_note_revision` with the job's revision number equal to the note's current one. The other note had been created by superseding an older one and had no root Bead; its job ended `planning_root_bead_missing`, which `planning_execution.py` marks non-retryable. A third cause in the same family is `schema_mismatch`, when the installed release does not support the job's input schema. In each case the explanation sat in the job's terminal reason, which the operator command of the time did not print. What changed: the eligibility classifier of 2026-10-13, the read-only status commands of 2026-11-08 that report the current job, its state and its reason, and a standing rule among the agents not to comment on a note whose job is queued or running. The lesson is to read the job, not the note, when a note has been in `planning` longer than the planner's deadline.

5.7. Approval Checks the Plan, Not the Approver

`note-approve` calls `approve` in `note_lifecycle_ops.py`. It requires a Lifecycle v2 note with an `active` disposition, and it runs the same evaluation the scheduler uses, `lifecycle_dispatch.evaluate_note`, ignoring only three guards that explicit approval is meant to override (`policy_deliver`, `auto_promotion_enabled`, `lifecycle_planned`). Then it sets the policy to `deliver` and the status to `open`.

The substance is in `_plan_evidence`, which runs thirteen named checks. Among them: `planned_revision_matches`, `planning_job_succeeded`, `planning_input_drift`, `plan_contract_valid`, `plan_fingerprint_matches` (the stored artifact hashes to the recorded fingerprint) and `validator_provenance_matches`. A refusal names every failed check.

What `approve` does not check is who is asking. Its `actor` parameter defaults to the string `operator`, and `note-approve` passes a label guessed from the calling environment (`_detect_source`), which is recorded and never compared with anything. Approval in code is proof that the plan is current and validator-backed. Whether the person or agent approving was entitled to do so is a matter of practice, and Chapter 18 (Signed Human Authority) describes that practice.

5.8. Arming Is Automatic

In the August edition, a dispatcher armed a planned note by hand. For Lifecycle v2 notes that is no longer the normal path. `yesod codefactory-dispatch arm` still exists. It refuses a note that is not `open` unless `--force` is passed (`check_arm_guard`), and forcing a new target onto a planned note before approval changes the input fingerprint, as described above.

A runner with `lifecycle_scheduler_enabled`, and not configured as an operator seat, runs `LifecycleScheduler.tick`. The tick selects v2 notes in `planned` or `open` with policy `deliver`, then locks and fully re-evaluates each one in its own transaction (`_reconcile_note`). A `planned` note that passes every promotion check is promoted to `open`, which is the `auto_promoter` authority at work for notes created with `--auto-plan`. A note that passes every dispatch check is armed.

Four fences on that path are worth naming. `lifecycle_rollout.scheduler_note_allowed` checks the rollout stage before the note is evaluated at all. Among the dispatch checks, the note's prerequisites must be ready, and for the scheduler only `done` satisfies a prerequisite (`SUCCESSFUL_PREREQUISITE_STATUSES`). Some active runner with a fresh heartbeat must advertise the exact target: the evaluation reports `no_live_advertiser`, and `require_live_advertiser` repeats the check when the arm is written. Last, `campaign_admission.evaluate_admission` applies tag-scoped capacity; a denial rolls the promotion back so the note is not left promoted and unarmed. The advertiser check dates from 2026-10-27 (`b342b82c9`); before it, the working notes record a note armed to a misspelled target that looked ready and that no runner would ever claim.

The target itself is chosen by `LifecycleDispatchPolicy.dispatch_resolution`. An explicit `agent_dispatch` already on the note is kept. Otherwise one fleet default is used: `DEFAULT_AUTO_DISPATCH_TARGET`, upgraded to `PREFERRED_GPT61_SOL_TARGET` only while a live runner advertises it. Nothing in this choice reads cost or complexity; Chapter 16 (Cost-Aware Dispatch) takes that up.

`_ensure_arm` then locks the tool's enabled refinery lanes and refuses unless there is exactly one. It computes `dispatch_idempotency_key` from the note id, revision, plan fingerprint, refinery lane, repository, target branch, source commit, dispatch target and execution placement, and inserts a row into `lifecycle_dispatch_arms` with `ON CONFLICT DO NOTHING`. The arm row is the durable authority. The `agent_dispatch` and `dispatch_idempotency_key` columns on the note are projections of it.

Approval and arming are separate transactions, so there is a gap between them. The working notes for 2026-10-11 record a runner that claimed inside that gap and failed with "no exact Lifecycle v2 dispatch arm". The arm arrived minutes later. The correct response was to wait.

5.9. Claimable Is a Table

`claimable_candidates` is a precomputed table, refreshed when notes change, when routed `yesod bd` commands mutate Beads, and by a periodic backstop (`claimable.py`). For a planned root, the row comes from one fresh evaluation of the Bead graph (`_planned_graph_candidate`),

which partitions the children into a terminal prefix, a current child and the suffixes behind it. If that read fails, `_record_observation_failure` stores the error beside the last good verdict and leaves that verdict alone; a note with no earlier row gets one with `verdict_authoritative` false. Before 2026-09-23 (`15fde3e01`) a failed read could leave a note unclaimable with no stated reason.

A runner may claim when two predicates hold. `ordinary_dispatch_candidate_sql` covers the note: status `open`, a dispatch target, a root Bead, an `active` disposition, the arm predicate `claim_arm_sql`, no active conflict repair, any retry delay elapsed, dependencies satisfied, and a candidate row that is both authoritative and ready. `runner_dispatch_admission_sql` covers the runner: a fresh heartbeat, spare capacity, the target in its advertised models, and pin rules. A separate predicate, `claim_not_drained_sql`, rejects a runner that is under a drain. The candidate table is only a hint. `claim_arm_sql` is applied again inside the atomic claim update, because a revision, refinery lane or policy change can race the read.

5.10. Holds and the Admissions Pause

Three things stop work in this part of the factory, and they are often confused.

A **hold** is a disposition. `note-hold` sets `held`, written through `write_disposition` and recorded in `note_disposition_changes`. It vetoes new planning leases, claims and merges. It does not cancel a planner that is already running.

"Hold every delivery" is operator policy. No such mechanism exists in the code on `main`. According to the agents' working notes, since 2026-10-13 every note that reaches `awaiting_merge` is held until a full test pass of its exact branch tip, run away from the merge queue, is clean. The policy was adopted after a 59-file change, 17 commits behind and never pre-run, was found two places from the head of the merge queue with 19 new failures. It is carried out by a polling script kept by the refinery clerk's seat, which calls the ordinary hold verb. The notes record its limits plainly: a poller cannot catch a delivery that is admitted between two polls on an idle queue, and on 2026-10-27 and 28 the pre-run step behind the hold stopped while the agent session that ran it was not running, so releases reached the merge queue untested. Chapter 17 (The Agent Organization) describes the seats; the point here is that this guard is a practice with a script behind it.

The **admissions pause** is code, and it is narrow. `require_admissions_open` in `refinery_admission.py` is called from refinery intake, the operator's branch-only submission, and the merge queue's enqueue and reconcile paths. It is not called from planning, arming or the runner's claim. While admissions are paused, notes are still planned, approved, armed, claimed and built; finished branches wait at `awaiting_merge`. Chapter 8 (The Merge Controller) owns the pause mechanism and its token.

Operator Check — Approved but not running. Use read-only commands only: `yesod tool <tool> note-why <id> --json`, `yesod tool <tool> note-plan-status <id>`, and `yesod planning-queue --human`. Read in this order. Disposition: anything but `active` stops everything. Policy: the scheduler acts only on `deliver`. Plan validity: `planning_input_drift` means something in the input moved after planning. Scheduler waits: `rollout_scheduler_fenced`, `campaign_capacity_exhausted`, `prerequisites_ready`, `refinery_lane_configured`, `no_live_advertiser`. Arm: if approval was minutes ago, wait for the next scheduler pass. Candidate row: `verdict_authoritative` and `ready`. Runner side: heartbeat, capacity, advertised targets, drains. Last, check that the runner machine is up at all; Chapter 6 gives the 2026-11-08 incident in which it was not.

5.11. What This Stage Guarantees

When a runner claims the note, the following is true and recorded: the plan was produced for the exact revision and input the note has now, a validator accepted it, an arm row fixes the refinery lane, branch, source commit and target, and the claim re-checked all of that in one transaction.

The stage does not establish that the plan is correct or that the approver was authorized, and the later hold before merge depends on a script that polls.

Source Notes

All code anchors are at `origin/main` `16606a42` (2026-11-17), paths under `yesod/src/yesod/`.

Claim or section	Tag	Anchor
Statuses, transitions, authority names	[code]	`note_lifecycle.py`: `IMPLEMENTATION_LIFECYCLE`, `CONTRACT`, `LifecycleDescriptor`
Authorities not compared with callers	[inferred]	grep for `authorities` across `yesod/src/yesod`; `write_status`
Policies and creation	[code]	`note_lifecycle_ops.py`: `policy_from_creation_flags`, `creation_values`
Planning request and re-plan	[code]	`request_planning`, `replan`, `ReplanRefused`; `tool.py note-plan`
Job identity, states, lease, attempts	[code]	`planning_jobs.py`; `planning_execution.py` `PlanningExecutionConfig`; `planner_fleet.py`
Input schema 3 and comments	[code]	`planning_input.py`; commit `ceae82033`
Plan contract and open questions	[code]	`contracts/planned_root.py`; `planning_contract.py`

Claim or section	Tag	Anchor
Open-questions loop figures	[notes]	agents' working notes, 2026-09-08 to 2026-09-11
Loop responses	[code]	commits `3fe504728`, `8bf282c70`
Material edit; stale revision	[code]	`invalidate_for_material_edit`; `planning_publication.py`
Dead planning incident	[notes]	agents' working notes, 2026-09-11; reason codes in code
Inspection commands	[code]	`docs/runbooks/planning-job-inspection.md`; commits `105a1b990`, `ee366b9ef`
Plan destroyed by `note-plan`; lost requirement	[notes]	agents' working notes, 2026-10-13 and 2026-11-08
Approval	[code]	`approve`; `lifecycle_dispatch.py` `evaluate_note`, `_plan_evidence`; `tool.py` `_run_lifecycle_operation`
Scheduler and arm	[code]	`LifecycleScheduler`, `_ensure_arm`, `dispatch_idempotency_key`; `runner.py` config flags; `check_arm_guard`
Target choice; live advertiser	[code]	`dispatch_target_resolution.py`; commit `b342b82c9`
Arm gap; misspelled target	[notes]	agents' working notes, 2026-10-11 and 2026-10-21
Claimability	[code]	`claimable.py`; `claim_arm_sql`; `claim_not_drained_sql`; commit `15fde3e01`
Hold every delivery	[notes]	agents' working notes, 2026-10-13, 2026-10-18, 2026-10-28
Admissions pause scope	[code]	`refinery_admission.py`; four call sites of `require_admissions_open`

Not verified:

- Whether any code path enforces a transition's named authority against the caller. A grep of the Python package found none; this is a negative search result, not a trace of every writer.
- The live rollout stage of Lifecycle v2 and the live planner concurrency limit. No database was read.
- The trace is constructed from code, not from a recorded note. Step 5 assumes a body edit is accepted while the disposition is `needs_rework`; this was not traced through the edit command.
- The scheduler counts only `done` as a satisfied prerequisite, while the runner's claim SQL (`_note_dependencies_satisfied_sql` in `claimable.py`) accepts `done` or `superseded`. The effect of the disagreement was not traced.
- Whether a dependency edit still invalidates a plan on the deployed release. `main` treats it as ordering-only; a working note of 2026-10-23 reports otherwise.

- Whether comment citations are accepted on the deployed release. `main` accepts them (`d1651abc5`); working notes of 2026-10-10 say citations had to come from the body.
- `renew_lease` in `planning_jobs.py` is called from `planning_execution.py`; the renewal interval was not traced.
- All counts in the two Factory Floor sidebars and in the hold policy paragraph are from the agents' working notes only.

6 The Runner

This chapter describes the daemon that turns an approved, armed note into one supervised attempt: how its loop is ordered, how it claims a note so that exactly one host owns it, how it runs agents without blocking, when it restarts itself, and which timers define "dead", "stuck" and "too long". It ends with the incident that taught the factory the most about this daemon: on 2026-09-10 its own self-healing watchdog was found to be killing the agents it supervised.

The reader should leave able to read a runner log and say which of three different heartbeats is missing, and why a runner that restarts itself needs a rule about when it may.

> **Terminology Trap — runner, dispatcher, lane.** The daemon is started by `yesod codefactory-dispatch` and implemented by the class `Runner` in `runner.py`. It is called the **runner** throughout this book. The dispatcher is an LLM role (Chapter 2, Judgment and Mechanism). A dispatch lane here means a model target such as `claude-opus`; the refinery's lanes (Chapter 8, The Merge Controller) are a different thing.

One runner process runs per runner host. It does five jobs: it keeps itself registered and heartbeating, it runs the durable planner and lifecycle ticks that Chapter 5 (Capture, Plan, Deliver) describes, it recovers work left behind by dead owners, it claims one eligible note at a time inside a PostgreSQL transaction, and it supervises the agent process it launches for that note.

6.1. Startup and the Wake

`Runner.start` registers the runner row, runs `_reconcile_startup_recovery`, and starts two background threads before it enters the loop: `_run_heartbeat_loop` and `_run_planning_harvest_loop`. It also starts two PostgreSQL listeners. One of them, `_run_arm_listener`, executes `LISTEN` on the channel `yesod_note_armed` and sets `_wake_event` when a notification arrives.

The loop itself is short. It calls `_run_poll_cycle`, then waits on `_wake_event` for at most `poll_interval` seconds (`DEFAULT_POLL_INTERVAL = 10`). The August edition called this a ten-second loop. Ten seconds is now only the upper bound between cycles: a NOTIFY sent when a note is armed wakes the loop at once.

6.2. One Poll Cycle

`_run_poll_cycle` calls fifteen steps in a fixed order. Grouped by purpose:

1. Diagnostics and a sandbox probe: `_tick_resource_diagnostics`, `_probe_bwrap_ready`.
2. Durable work that lives in PostgreSQL: `_tick_planning`, `_tick_lifecycle_scheduler`, `_tick_remote_recovery`.
3. Recovery of other owners' work: `_release_dead_runner_work`, `_reap_stalled_notes`.
4. Claiming: `_probe_bd_ready`, then `_poll_and_process`.
5. Maintenance: `_tick_gate_maintenance`, `_tick_test_schema_sweep`, `_tick_workspace_hygiene`.
6. Self-assessment: `_evaluate_operational_health`, `_tick_dispatch_watchdog`, `_tick_readiness_re`

The order is deliberate, and the method's docstring gives the reason. The planner and lifecycle ticks come first because the Beads readiness, reaper and maintenance paths shell out to other programs and can degrade; they "must never starve durable jobs while the daemon continues to emit a healthy heartbeat".

Recovery has two parts. `_release_dead_runner_work` marks a peer dead when its row heartbeat is older than `dead_threshold` (`DEFAULT_DEAD_THRESHOLD = 300` seconds), but before it reopens any of that peer's notes it checks the run ledger for a live, heartbeating run. A daemon whose heartbeat thread was starved must not lose a note whose agent is still working. `_reap_stalled_notes` runs every `DEFAULT_REAPER_INTERVAL = 60` seconds and is scoped to the runner's own claims: it returns to `open` a note this runner claimed that has sat for `DEFAULT_REAPER_THRESHOLD = 300` seconds in `in_progress`, or in `awaiting_verification` with no recorded reason, with no live heartbeating run in the ledger.

`_evaluate_operational_health` can set the runner `degraded` after `DEFAULT_DEGRADE_FAILURE_THRESHOLD = 5` consecutive failed operational checks. A degraded runner finishes what it holds and claims nothing new.

6.3. The Claim

Take the book's running example: the note that asks for a read-only command explaining why an armed note is not running. After approval and arming (Chapter 5) the note has status `open`, a live arm, a linked root bead, and three planned child beads of which the first is ready.

`_claim_note` finds candidates by intersecting two durable inputs: armed notes in PostgreSQL (read through the `claimable_candidates` read model where it has rows) and dependency-ready beads from `bd ready`. A ready child bead is resolved to its root, and the dispatch note is the one whose `bead_id` is that root. Several ready children of one root collapse to one candidate, so the runner claims the root note once and runs its children in sequence.

Every `bd` and `yesod` subprocess call happens outside any open database transaction. The docstring records why: holding a transaction while those subprocesses opened their own connections caused "lock/connection contention and 30s timeouts".

The claim transaction then does the following, in order:

1. Locks the note row (`_note_for_bead` with `FOR UPDATE OF n`) and abandons the claim if the note no longer matches. Notes are always locked before runners, so two claim transactions cannot wait on each other in a cycle.

2. Takes a transaction-scoped advisory lock keyed by `claimable.runner_drain_lock_key(runner_id)`, then locks the runner row with `SELECT current_tasks ... FOR NO KEY UPDATE` and rechecks capacity.
3. Abandons the claim if a live row for this runner exists in `release_promotion_drains` or `routine_rollout_drains`. A rollout that wants this host empty takes the same advisory lock before it writes its drain row, so the two transactions are ordered.
4. Abandons the claim if this runner already has a live owning run (`_claim_blocked_by_live_runner_r`
5. Updates the note to `claimed` with `WHERE note_id = ... AND status = 'open'` plus the arm and drain predicates. If no row is updated, another runner won the race.
6. Inserts a provisional run row in status `starting` (`_insert_starting_run_for_claim`, which calls `run_ledger.insert_starting_run`), when the runner has found the `runs` schema capable of it.
7. Increments `runners.current_tasks` and records the `open` → `claimed` transition.

Only `psycopg.errors.DeadlockDetected` is retried, by `_with_deadlock_retry`. The bead is assigned afterwards by `_assign_bead`, outside the transaction; if that fails the runner compensates by releasing the claim. PostgreSQL and the Beads database cannot commit together, so the design is a short authoritative claim followed by an explicit repair path.

> **Invariant — A claim always has a run row.** Since commit `98f3b8262` (2026-10-13) the `starting` run row is inserted in the same transaction that sets the note `claimed`. A note is then never owned by a runner with nothing in the ledger to show for it. `promote_starting_run` later moves the row to `running`. The one exception is deliberate: against a database that predates the migration, `_claim_with_starting_run_fallback` retries the claim without the run row, so that a runner rolled ahead of the schema still dispatches.

> **Factory Floor — The runner waited on itself (2026-09-25).** The claim transaction used to lock its runner row `FOR UPDATE`. Its bail-out paths write a line to `runner_logs` through a second connection, and that insert needs a key-share lock on the same runner row, which `FOR UPDATE` blocks. The claim waited on its own log line; two gate tests hung. Commit `10090d261` (03:43Z that day) changed the lock to `FOR NO KEY UPDATE`, which still serializes the capacity check and no longer blocks a foreign-key reference to the row.

6.4. The Worker Pool

`_poll_and_process` never blocks on an agent. It reaps finished workers, returns at once if the runner is degraded, and otherwise loops: count live workers, stop if the count has reached `max_concurrent`, claim a note, and hand it to `_start_worker`. Each claimed note runs in its own thread named `task-worker-<note_id>`, and that thread owns the whole attempt: worktree, agent launch, log tailer, ledger row, the governed wait, and completion or failure handling. `DEFAULT_MAX_CONCURRENT = 1`, so by default a runner holds one note. Capacity is checked twice, against the live thread count and again inside the claim transaction against `runners.current_tasks`.

A runner has two ways to restart itself and one way to stop.

RSS recycle. When no worker is live, `_poll_and_process` calls `_recycle_high_rss_runner` before claiming. It reads the process's anonymous resident memory, and above `max_rss_recycle_mb`

(DEFAULT_MAX_RSS_RECYCLE_MB = 2048) it stops the loop and exits with RUNNER_SERVICE_RESTART_EXIT_COD = 75, so that a service unit configured to restart on failure starts a fresh process. A value of 0 disables it, and an unreadable reading skips the check with a warning. The restart can only happen between claims, when the runner holds nothing.

Watchdog restart. _tick_dispatch_watchdog restarts a runner whose view of ready work has gone stale. It is the subject of the incident below.

Shutdown drain. SIGTERM only sets a flag. _shutdown_graceful calls _drain_workers, which waits up to shutdown_drain_seconds (DEFAULT_SHUTDOWN_DRAIN_SECONDS = 300.0) for in-flight workers, then terminates the survivors' process trees and allows 30 more seconds for their final writes. The runner row becomes inactive only if run_session_ledger.list_runner_retiremen reports that transcript archival is complete; otherwise it is left degraded with an error. The release sweep returns to open only notes still in claimed or in_progress. Notes that reached a later status keep it.

6.5. Worktrees and the Accumulating Branch

All agent work happens in Git worktrees under DEFAULT_WORKTREE_BASE_DIR, which is /tmp/yesod-workdir, outside any repository tree. The worktree name and the default branch name are both <tool>-<note_id>. Operations on the shared cached clone are serialized by _git_lock.

A failed git fetch origin --prune produces no worktree and an error log line beginning source_refresh_failed. Cached remote refs are not accepted as evidence of current source (commit 75e42fcfe, 2026-09-16).

For a planned note the runner executes children one at a time on one branch. The first child's worktree is cut from origin/main or the note's target branch. After that child's commit is pushed, the second child's worktree is cut from origin/<tool>-<note_id>, and so on. This is the **accumulating branch**: one branch per note, each child starting from its predecessor's pushed result. In the example, the three children of the diagnostic-command note produce three or more commits on one branch, and that branch is what the refinery later receives.

On the success path _cleanup_worktree pushes and then removes the worktree. Failed and unverified worktrees are kept for triage, and a worktree whose push failed is kept as well. Workspace hygiene runs inside the worktree-create path before each run, and _tick_workspace_hygiene repeats it over every cached clone hourly (DEFAULT_WORKSPACE_HYGIENE_INTERV = 3600.0), to repair damaged clones and stale registrations before they block a claim.

6.6. Timers That Define Failure

An agent can hang without crashing, so the limits are constants in runner.py, most of them overridable in the runner's configuration file.

Control	Constant	Default
Poll upper bound	`DEFAULT_POLL_INTERVAL`	10 s
Runner row heartbeat	`DEFAULT_HEARTBEAT_INTERVAL`	30 s
Dead runner	`DEFAULT_DEAD_THRESHOLD`	300 s
Loop liveness bound	`DEFAULT_LOOP_LIVENESS_BOUND`	600 s
Stalled-note reaper	`DEFAULT_REAPER_THRESHOLD`	300 s
Soft host affinity	`DEFAULT_SOFT_AFFINITY_TIMEOUT`	300 s
Agent wall clock	`DEFAULT_TASK_TIMEOUT`	1,800 s
Governance poll	`DEFAULT_GOVERNANCE_POLL_INTERVAL`	5 s
Token sample	`DEFAULT_TOKEN_SAMPLE_INTERVAL`	30 s
TERM to KILL grace	`DEFAULT_TERM_GRACE_SECONDS`	30 s
Shutdown drain	`DEFAULT_SHUTDOWN_DRAIN_SECONDS`	300 s
RSS recycle	`DEFAULT_MAX_RSS_RECYCLE_MB`	2,048 MiB
Watchdog threshold	`DEFAULT_DISPATCH_STUCK_THRESHOLD`	30 cycles
Watchdog cooldown	`DEFAULT_DISPATCH_STUCK_RESTART_COOLDOWN`	900 s
Push retries	`DEFAULT_BRANCH_PUSH_RETRIES`	3, 2 s apart
Execution retries	`DEFAULT_MAX_RETRIES`	3

Two of these interacted badly in the incident that ends this chapter: a watchdog that fires after about five minutes, and a drain that kills five minutes after that.

6.7. The Run Ledger and Three Heartbeats

Every claim opens a row in the `runs` table. `_run_start_parameters` fixes its identity at claim time: note, bead, runner, host, dispatch lane (stored in the `agent` column), model, the wall-clock limit in `max_runtime_seconds`, whether a spend limit can be enforced, and the release SHA the runner is executing. A run is live in status `starting` or `running` (`LIVE_RUN_STATUSES`) and ends in one of `success`, `failure`, `killed`, `timeout`, `lost` or `token_budget` (`TERMINAL_STATUSES`). `reconcile_lost_runs` settles rows left live by a dead daemon to `lost`.

The note records the life of the requested change. The ledger records each attempt that spent time and money on it. A note that lands on its third run still cost three runs.

"The heartbeat" is ambiguous on a runner, because there are three:

- **The runner row.** `_run_heartbeat_loop` updates the `runners` row every 30 seconds on its own thread. It stops beating if the main loop has not advanced for `DEFAULT_LOOP_LIVENESS_BOUND = 600` seconds, so that a hung loop looks dead to its peers.
- **Agent progress.** The agent calls `yesod dispatch-agent-log <note_id> "<summary>"`. Each call appends a timestamped line to the per-note log file and inserts a row in

`dispatch_log`. The module's own docstring calls this signal "self-reported, prompt-dependent, lossy". Failures of the command never fail the task.

- **The runner-side run heartbeat.** While an agent is alive and silent, the governed wait writes a `dispatch_log` row reading "runner heartbeat — agent/gate active" every heartbeat interval, so a long quiet test run does not look dead to the reaper.

Planning work has its own liveness signal besides these, a heartbeat on the planning publication record; which thread writes it was not traced for this chapter. The working notes for 2026-09-12 describe a runner whose row was `dead` while its planning publication was fresh and its work finished. A report that "the heartbeat is stale" is incomplete until it says which one.

> **Operator Check — Is anything alive to claim?** Before reasoning about why a note is not running, list the runner rows, including dead and degraded ones, and the live runs with `yesod runs list`. The working notes for 2026-11-08 record two agents reasoning for hours from job-state snapshots about a "frozen planning queue" while the runner VM that owned the work had been stopped for more than fifteen hours.

6.8. The Watchdog That Killed the Work

The dispatch watchdog exists for a real fault. A runner can pass every probe and still serve a stale readiness view: armed notes whose beads are ready never appear in its ready set, every cycle logs a skip, and nothing is claimed until someone restarts the daemon. `_tick_dispatch_watchdog` counts consecutive "starved" cycles, meaning cycles in which the runner has spare capacity and at least one armed candidate is missing from the ready set. After `DEFAULT_DISPATCH_STUCK_THRESHOLD = 30` such cycles, about five minutes at the ten-second poll, `_trigger_dispatch_self_heal` logs "dispatch watchdog FIRED", requests a graceful stop, and the process re-executes itself.

Factory Floor — The self-heal that destroyed work (found 2026-09-10). Agents had been dying with SIGTERM in the middle of planned children, and the deaths had first been blamed on RSS recycling. According to the agents' working notes for 2026-09-10, the cause was the watchdog. Its starvation test compared a note's root bead against a ready set that contains children, so for a planned note whose work is carried by children, "armed but never claimed" was permanently true. The counter climbed to the threshold, the watchdog fired, the runner began a graceful restart, and when the 300-second shutdown drain expired it terminated whatever agent was still running. The notes record 36 watchdog fires that day, and 17 drain-timeout kills, every one of them 300 to 311 seconds after a fire on the same runner. They estimate 979 agent-minutes destroyed against 2,346 productive. These counts come from one agent's reading of that day's runner logs and were not re-measured for this book. The other fires landed on idle runners and left no trace, which is why the pattern stayed invisible.

The notes add two details worth keeping. The defect had been filed as a note twenty-one days earlier and had never been planned. And an attempt to mitigate it that evening by raising the drain in the runner's configuration file did nothing, because the configuration is read once at process start; the next kill came about 288 seconds after its fire.

Two changes followed, both verified on `main`. Commit `6330a83aa` (2026-09-10) makes `_trigger_dispatch_self_heal` check `_live_worker_count()` and, when it is not zero, log "auto-restart suppressed because the runner is busy" and return. Commit `8e7f28570` (2026-09-15) makes the starvation test require positive canonical readiness before it counts a cycle as starved.

The rule now enforced is that a busy runner is not restarted by its own watchdog: `_trigger_dispatch_self_heal` returns without restarting when any implementation worker is live. RSS recycling obeys the same rule from the other side, because `_poll_and_process` calls it only when the live worker count is zero.

The general lesson is about the order of fixes. A self-heal that restarts a process holding work can cost more than the fault it heals. The first rule is that the remedy must not touch in-flight work; correcting the predicate comes second; tuning the timers comes last, because a longer drain would have left three dozen spurious restarts a day in place.

With the note claimed, the run row open and the worktree cut, the runner starts the agent. Chapter 7 (Governed Agent Execution) covers what that agent is told, what it is allowed to see, and what the runner demands before it believes the work is done.

Source Notes

Source snapshot: `yesod-aicoe origin/main` at `16606a42` (2026-11-17). Paths are under `yesod/src/yesod/`.

Claim or section	Tag	Anchor
Startup and the wake	[code]	`runner.py`: `Runner.start`, `_run_arm_listener`, `_wake_event`, `DEFAULT_POLL_INTERVAL`

Claim or section	Tag	Anchor
Poll cycle order	[code]	`runner.py`: `_run_poll_cycle` and its docstring
Dead-runner and stalled-note recovery	[code]	`runner.py`: `_release_dead_runner_work`, `_reap_stalled_notes`
Claim transaction	[code]	`runner.py`: `_claim_note`, `_with_deadlock_retry`, `_assign_bead`; the two drain tables
Run row with the claim	[code]	`run_ledger.insert_starting_run`, `promote_starting_run`; commit `98f3b8262`
Runner waited on itself	[code]	commit `10090d261` (2026-09-25 UTC), commit body
Worker pool, RSS recycle	[code]	`runner.py`: `_poll_and_process`, `_start_worker`, `_recycle_high_rss_runner`; commit `c0df0edbf`
Shutdown drain	[code]	`runner.py`: `_shutdown_graceful`, `_drain_workers`
Worktrees, accumulating branch	[code]	`runner.py`: `_create_worktree_unlocked`, `_cleanup_worktree`, `_git_lock`; commit `75e42fcfe`
Timer table	[code]	constants of the same names in `runner.py`
Run ledger	[code]	`runner.py`: `_run_start_parameters`; `run_ledger.py`: `LIVE_RUN_STATUSES`, `TERMINAL_STATUSES`
Three heartbeats	[code]	`runner.py`: `_run_heartbeat_loop`, `_wait_with_governance`; `dispatch_agent_log.py`
Dead row, live planning work	[notes]	agents' working notes, 2026-09-12
Stopped runner VM read as frozen queue	[notes]	agents' working notes, 2026-11-08
Watchdog mechanism and fixes	[code]	`runner.py`: `_tick_dispatch_watchdog`, `_trigger_dispatch_self_heal`; commits `6330a83aa`, `8e7f28570`
Watchdog cause, counts, failed mitigation	[notes]	agents' working notes, 2026-09-10

Not verified:

- The watchdog incident's counts (36 fires, 17 kills, 979 and 2,346 agent-minutes), the root-versus-child cause, the twenty-one-day-old note and the 288-second kill are from the agents' working notes only. The two code fixes are verified.
- The size and composition of the live runner fleet on 2026-11-19, and each runner's configured `max_concurrent` and drain values. Source gives defaults only.
- That the service unit on production runners restarts on exit code 75. The code comment says the deployed unit is `Restart=on-failure`; the unit file was not read for this chapter.

- The planning liveness signal is described from the notes (a publish-intent heartbeat) and from the existence of `_run_planning_harvest_loop`; which code writes it was not traced.

7 Governed Agent Execution

The runner has claimed the note and cut a worktree. This chapter follows the coding agent that now runs inside it: what instructions it receives, which environment variables it can see, what limits surround its process, and what the runner requires before it treats the work as done. It then covers the two ways a planned child may end without the usual proof, and the blind spot the second one opened: as of 2026-11-19 a child task can commit, close, and travel toward merge without any of its prescribed tests having run.

The principle is the same as in August. The agent has wide freedom to read, edit, test and commit. It has no authority to decide that a missing commit is close enough, to extend its own deadline, or to merge. What changed is how much of that boundary is now a typed record and how much is still an instruction in a prompt. The chapter says which is which.

7.1. The Worker Canon

A dispatched worker's instructions are generated from one module, `worker_canon.py`. Its `SECTIONS` mapping holds eleven sections in render order: `command-crib`, `workflow`, `tooling-standards`, `gate-test-guidance`, `test-database-hygiene`, `scratch-vm-policy`, `heartbeat`, `git-discipline`, `worktree-conventions`, `root-bead-ownership` and `write-habit`. Chapter 4 (The Registry Beneath the Factory) describes the four consumers that render from it, of which the dispatched prompt (`_build_agent_prompt` in `runner.py`) is the one that matters here, and the `SENTINELS` test that keeps a safety rule from being edited in one copy and forgotten in another.

The rules that matter most, each with the incident or reason the text itself gives:

- **Git discipline.** The worktree is already on the right branch. The canon forbids `git reset`, `rebase`, `pull`, `merge`, switching branches, force pushes, `reflog expire` and `gc --prune`, and sums up: "Never move HEAD except by committing." An agent that "syncs with main" moves its branch off its own commits, and the runner then finds no work.
- **Commit incrementally, and before any long validation.** A run can be killed at any moment, and an uncommitted worktree is lost. The agents' working notes for 2026-09-20 record two unattended runs, started by an operator-side launcher and not by the runner, that parked a gate poll in a background task, ended their turn to wait, and were cut off by the harness after 600 seconds with a tidy summary and the task unfinished. The canon says "COMMIT BEFORE ANY LONG OR BACKGROUND VALIDATION".

- **Targeted tests, never the full suite.** The refinery runs the complete gate at merge time (Chapter 9, The Deterministic Gate). A full local run "regularly consumes your whole time budget on one slow pass", in the canon's words.
- **Test-database hygiene.** Added on 2026-10-14 (commit `9666f0b85`). The test suite's `conftest.py` resolves the ambient database setting and creates or reuses a shared gate database before any test body runs, so even a test that looks database-free touches shared state. An ordinary worker therefore must not execute pytest at all unless it has an explicitly worker-local test database. It runs `uv run pytest tests/... --collect-only -q` to prove the tests import and collect, and leaves execution to a later stage.
- **Never close the root bead**, route every Beads command through `yesod bd`, narrate progress with `yesod dispatch-agent-log`, and file durable facts back to the registry.

The hygiene rule is the right response to a real hazard. It is also one half of the blind spot described later in this chapter.

7.1.1. Plan-Bound Leaves

For a planned note the runner selects a stricter variant. `_PLAN_BOUND_LEAF_SECTIONS` replaces three sections for a child bead that the planning validator published. Such a child is a plan-bound leaf (Chapter 3, The Two Currencies): the agent works directly in the assigned bead, keeps any checklist local, and must not create descendant beads, claim or close siblings, or close the leaf itself. The runner owns progression among siblings. A change to the graph requires formal replanning.

This rule has a hard edge in code as well as in the prompt. The routed Beads adapter in `beads.py` refuses to create a descendant under a plan-bound leaf (error code `plan_bound_leaf_descendant`) and refuses an unscoped atomic claim (`plan_bound_leaf_atomic_claim_scope_forbidden`). The leaf's context reaches the agent in the environment variable `YESOD_PLAN_BOUND_LEAF_CONTEXT`. This is why the canon insists on `yesod bd` and never raw `bd`: the guard lives in the routing.

7.2. The Launch Environment

The runner daemon holds many credentials. The agent it launches is treated as untrusted and receives a fresh environment built by `build_agent_launch_environment` in `agent_launch_environment.py` (introduced 2026-10-11, commit `c62e08e41`). The policy is deny by default. Unknown ambient variables are never copied.

Layer	Constant	Contents
Common	`COMMON_ENVIRONMENT_KEYS`	`PATH`, `HOME`, `LANG`, `TERM`
Provider	`PROVIDER_ENVIRONMENT_KEYS`	model credentials for one of `codex`, `claude`, `opencode`
Yesod CLI	`YESOD_CLI_ENVIRONMENT_KEYS`	settings the `yesod` command needs

Layer	Constant	Contents
Run overlay	WORKER_LAUNCH_OVERLAY_KEYS	run id, note id, leaf context, archive ids

Values in the overlay are generated by the launcher; an overlay key outside the allowed set raises an error, and any key beginning with a prefix in FORBIDDEN_AGENT_ENVIRONMENT_PREFIXES (the cloud provider's credential variables) is rejected outright. A worker receives only the key set declared for its own provider family. The sets overlap, and the opencode set is the widest: it names keys for several model vendors.

The database credential is the honest weak point. The design gives each runner a separate least-privilege credential for agents, which the builder renames to the variable the yesod CLI reads. When that scoped credential is not configured, _map_postgres_capability passes the daemon's own credential and emits DAEMON_DSN_FALLBACK_WARNING, which begins "SECURITY WARNING". The runbook docs/security/agent-launch-environment.md calls this a compatibility fallback for the rollout window. Whether every production runner has the scoped credential today is not established from source.

The prompt itself is also handled as data. On main since 2026-10-28 (commits 514b2f479 and 758a04a5a) Codex prompts are delivered over standard input instead of the command line (prompt_transport.py), in processes whose configuration came from RunnerConfig.load(). The change needs a drained fleet rollout, described in a runbook; whether that rollout has completed on every live runner was not checked. On the argument-only path, a prompt at or above ARGV_PROMPT_LIMIT_BYTES (120 KiB) is refused with reason code prompt_exceeds_argv_limit, and _park_prompt_launch_failure parks the note instead of retrying a launch that will fail the same way every time.

7.3. Limits Around the Process

The agent runs in its own process session. The worker thread waits in _wait_with_governance, which returns one of four outcomes:

- exited: the agent finished on its own.
- timeout: the wall-clock limit from _effective_max_runtime passed. The limit resolves from an override for the dispatch lane, then the global setting, then the legacy task_timeout, then DEFAULT_TASK_TIMEOUT = 1800 seconds.
- token_budget: cumulative input tokens, sampled every DEFAULT_TOKEN_SAMPLE_INTERVAL = 30.0 seconds, crossed max_input_tokens. This backstop is armed only when a budget is configured. The comment on the constant records the runaway that motivated it, about 52 million tokens in an hour.
- killed: a kill request for this run was found in the ledger.

In the last three cases _terminate_process_tree sends SIGTERM to the whole process group, waits DEFAULT_TERM_GRACE_SECONDS = 30.0, and then sends SIGKILL. A failed kill poll or a failed token sample keeps the run alive; a database blip must not end an agent.

> **Proven Where — runtime and spend limits, as of 2026-11-19.** On `main`, the wall-clock limit is enforced in exactly one place: the owning runner's wait loop, once per agent process. No other component kills a run that outlives its recorded limit. `run_ledger.request_kill` has a single caller on `main`, the `yesod runs kill` command in `runs_cli.py`, which a person or an agent must choose to run, and it marks only rows in status `running`. `fleet_status.py` labels an overlong run with no commits `over_cap` for display and takes no action. The agents' working notes for 2026-09-09 describe the limit as recorded and never enforced at that date: they record one run that reached 5 hours 6 minutes against a two-hour limit, and none of thirteen runs killed in 24 hours. No commit on `main` names the note filed about it. One reading consistent with the source, inferred and not traced: a planned note has one run row carrying one `max_runtime_seconds`, while `_run_child_agent` starts a fresh deadline for each child, so the run as a whole is bounded only by the number of children. Spend is weaker still. When a spend limit is configured, `_spend_enforcement` records `not-enforceable` on the run, because no backend sets `cost_observable`; the docstring says "none do today".

Progress events are a separate matter. `yesod dispatch-agent-log` lines tell an operator what the agent believes it is doing. They are not proof of anything. A run can narrate well and commit nothing.

7.4. Proof of Work

When the agent exits, the runner evaluates artifacts.

For an ordinary, unplanned note, `_handle_success` calls `_verify_work_complete`, which requires commits past the recorded base. With none, the note goes to `awaiting_verification` with disposition `needs_rework` and a reason beginning `no_commits:`, or `branch_reset:` when `_detect_orphaned_commits` finds commits the agent abandoned by moving its branch. A note with `task_kind=data` replaces the commit test with its acceptance probe, capped at `ACCEPTANCE_CHECK_TIMEOUT_SECONDS = 120`. If the agent closed the root bead, the runner reopens it. `_push_and_verify_branch` then pushes (with `--force-with-lease` on the branch ref when it is given an explicit branch), retries a failed push up to `DEFAULT_BRANCH_PUSH_RETRIES = 3` more times, and confirms the remote ref with `git ls-remote`. A local commit is not enough, because the refinery runs on another machine.

For a planned note the check is per child and stricter. Follow the example note from Chapter 6 (The Runner) with its three children. `_process_planned_root_dispatch` observes the Beads graph, claims the first child atomically, records immutable events (`graph_observation`, `claim_outcome`, `launch_boundary`), and runs the agent through `_run_child_agent`. When the agent exits, `validate_leaf_result_commit` in `root_dispatch_reconciliation.py` requires four things: a captured base SHA; a full result SHA equal to the worktree's exact `HEAD`; the base as an ancestor of the result; and at least one commit between them. The branch tip is pushed and verified. Then `reconcile_successful_child` closes the child bead with a canonical reason and reads the bead back to confirm it is terminal:

```
yesod-root-dispatch-child:v1;note_id=<id>;
  child_bead_id=<id>;result_commit=<40 hex>
```

(The reason is one line; it is wrapped here.) Only then does the loop look at the graph again and start the second child from the pushed branch.

> **Invariant — A sibling never starts on an unknown predecessor.** `_run_child_agent` reports success only after `reconcile_successful_child` has closed and re-read the child. A failure at any step parks the note in `awaiting_verification` with reason code `root_dispatch_child_reconciliation_failed` and one of the codes in `FAILURE_CODES`, such as `result_commit_missing`, `result_commit_not_leaf_local` or `child_not_terminal_after_close`.

The agent does not close its own leaf. The runner closes it, and the close reason carries the commit.

7.5. Ending Without a Commit

Some children legitimately produce no commit. Since 2026-09-06 (commit `e17b48214`) there is a typed way to say so. The agent emits exactly one heartbeat whose summary is the prefix `YESOD_CHILD_OUTCOME:` followed by one JSON object, and the runner reads it from the event store. `ChildOutcomeKind` now has three values, the third added on 2026-10-13 (commit `de8ca8eaa`):

- `blocked_on_dependency`, naming a real note, branch or bead and a reason.
- `no_changes_needed`, with at least one check (command, directory, exit status, a summary of at most 200 characters) and a one-line assertion of at most 400 characters saying what already satisfies the task.
- `acceptance_not_executed`, with one of four reasons. This one is different: it is required even when the child did commit.

```
$ yesod dispatch-agent-log ys-yes-abcd \
    'YESOD_CHILD_OUTCOME:{"outcome":"acceptance_not_executed",
    "reason":"db_env_unset"}'
```

(The payload is one compact JSON object; it is wrapped here, and the note id is a placeholder.) Two prefixed heartbeats in one attempt are a contract violation even when identical (`select_child_outcome_heartbeat`). Prose such as “blocked, aborting” is not an outcome. A child with no commit and no outcome is `child_outcome_missing`: `_persist_missing_child_outcome` reopens the child bead and gives the note the disposition `needs_rework`. A plan in which every child proved `no_changes_needed` ends through `_finish_completed_no_change_plan` with nothing pushed.

7.6. The Blind Spot: Committed, Closed, Untested

On 2026-10-13 a second structured record was added (commits `de8ca8eaa` and `344dd8b73`). For every acceptance command the planner published for a child, the worker must emit one

`YESOD_ACCEPTANCE_EVIDENCE:` record with exactly eight fields: `command`, `cwd`, `exit_status`, `collected`, `passed`, `failed`, `skipped` and `db_env_present`. `classify_child_acceptance` in `acceptance_evidence.py` turns the records into one of three dispositions: `verified`, `deferred_to_prerun` or `blocked`.

Now put the pieces together for the third child in the example, whose acceptance command is a database-backed test.

1. The hygiene rule forbids the worker to run pytest without a worker-local test database.
2. The runner can create such a database per run, but only when `disposable_postgres_enabled` is true. `RunnerConfig` defaults it to `False`, and the runner birth scripts on `main` write it as `false` explicitly.
3. The worker therefore records the command as refused, with `db_env_present` false, commits its code, and emits `acceptance_not_executed` with reason `db_env_unset`. The canon says this outcome "means deferred-to-prerun, never Completed".
4. The runner closes the child with the deferred form of the close reason, which appends `outcome=acceptance_not_executed`, `reason=db_env_unset` and an evidence digest to the result commit (`deferred_child_close_reason`).
5. At the root, `_finish_planned_root_dispatch` moves the note to `awaiting_merge` with verification reason `acceptance_deferred_to_prerun` and enqueues it for the refinery with an `unverified` flag. It would stop instead only if `acceptance_hard_block_enabled` were true. That setting also defaults to `False` and is also written `false` by the birth scripts.

Every step is honest. The record says, in a machine-readable field, that no test ran. Nothing in the default configuration acts on it.

> **Factory Floor — "All nineteen children closed" (2026-11-19).** The hardening work called H1 was delivered as one note with nineteen child tasks. According to the agents' working notes for 2026-11-19, every child's close reason read `outcome=acceptance_not_executed;reason=db_env_unset`, and a deterministically failing new test was then found in the delivered tip by running the tests separately. The children had closed, the commits existed, and no prescribed test had executed on the build seat. The ops README on `main` describes turning on disposable PostgreSQL and the hard block as "separate reviewed rebirth rollouts"; neither had happened for the seats involved, as far as the notes record. The practical rule adopted the same day: a closed child is evidence of a commit, and the close reason says whether it is evidence of anything more.

Chapter 15 (The Reliability Contracts) takes up what this does to the durable-completion contract, which until November read as though a pushed commit and a closed child were sufficient proof.

7.7. Execution Providers, and What Is Dormant

Every launch goes through `ExecutionBroker` in `execution_broker.py` and returns an `ExecutionHandle`. `execution_placement.py` knows two placements, `local` and `lambda-microvm` (`KNOWN_EXECUTION_PLACEMENTS`), and the default policy allows only `local`. The placement is bound to the arm and the run with a digest of the effective configuration, so a run cannot drift

to another placement after approval. `execution_provider.py` defines the provider-neutral request and receipt, and `lambda_microvm_execution.py` is an adapter written against it.

> **Proven Where — execution placement, as of 2026-11-19.** In production, agents run as local processes on runner VMs. The Lambda MicroVM adapter is on `main`; its own architecture document says it is "conditionally feasible at the adapter seam and not yet eligible for live delivery", reviewed offline on 2026-08-25. No evidence on `main` shows a production run placed anywhere but `local`. The inference gateway boundary (`inference_gateway/`) and the hosted execution path are likewise code on `main`; whether the per-run gateway lease is switched on for the live fleet is not established. Scratch machines (`yesod scratch`), short-lived VMs leased to agents under `docs/policies/scratch-vms.md`, exist for work that needs a disposable host.

7.8. Failure on Two Axes, and Bounded Retries

A failed attempt is described twice, by two modules; the docstring of `failure_analysis.py` calls the two classifications orthogonal.

`reconciliation.py` says what happened to the work. Its verdicts are `work-complete-unclaimed`, `work-present-uncommitted`, `partial-progress`, `no-progress`, `infra` and `needs-reconcile`. `failure_analysis.py` says why the attempt ended: `over-explore`, `runaway`, `fast-fail`, `gate-hang`, `no-op`, `lost-work`, `timeout`, `token-budget`, `killed`, `missing-logs` or `unknown`. "No commit" can mean the model did nothing, a tool failed before launch, good edits sit uncommitted, or the behavior already existed. A retry policy that looked only at the exit code would treat those alike.

`_handle_failure` increments the note's `retry_count` and then chooses:

- If this attempt and the previous one both reconciled to `no-progress`, it stops at once, whatever allowance remains. `_write_failure_note_state` sets the disposition `blocked` and asks the status writer for the older status of the same name. The retry disposition recorded for the run is `retry_halted_no_progress`.
- If the count has reached `max_retries` (`DEFAULT_MAX_RETRIES = 3`), it blocks the note and records `retry_exhausted`.
- Otherwise the note returns to `open` with a backoff of `2 ** retries` minutes, and `_maybe_escalate_dispatch` may move the next attempt to `ESCALATION_DISPATCH_TARGET`, which is `claude-opus`, when the verdict points at model ability. Verdict `infra` never escalates.

Rate-limit backoff (`_handle_rate_limit`) and claim collisions (`_handle_claim_collision`) are handled on their own paths. A planned child interrupted by a runner restart is re-armed automatically, with at most `INTERRUPTED_CHILD_MAX_LAUNCHES = 3` launches counted from the immutable launch events.

> **Design Trade-off — Retry only when the next attempt differs.** Backoff, a stronger dispatch lane, a repaired dependency or a triaged fix each change the experiment. Two attempts with zero delta are evidence that a third identical one will also produce nothing, so the guard pays for two and stops.

A successful run ends with a verified remote branch and a note in `awaiting_merge`. In current practice the note is then held before the refinery takes it (Chapter 5, Capture, Plan, Deliver); what the refinery does with it is the subject of Chapter 8 (The Merge Controller).

Source Notes

Source snapshot: `yesod-aicoe` `origin/main` at `16606a42` (2026-11-17). Paths are under `yesod/src/yesod/` unless they begin with `docs/` or `ops/`.

Claim or section	Tag	Anchor
Worker canon rules	[code]	`worker_canon.py`: `SECTIONS`, `SENTINELS`; commit `9666f0b85`
Background-task cutoffs behind the commit rule	[notes]	agents' working notes, 2026-09-20; rule text in `_GIT_DISCIPLINE`
Plan-bound leaves	[code]	`worker_canon.py`: `_PLAN_BOUND_LEAF_SECTIONS`; `beads.py` error codes; commit `8ab2420fe`
Launch environment	[code]	`agent_launch_environment.py`: `build_agent_launch_environment`, `_map_postgres_capability`; `docs/security/agent-launch-environment.md`
Prompt transport	[code]	`prompt_transport.py`; `ARGV_PROMPT_LIMIT_BYTES`; the stdin rollout runbook; commits `514b2f479`, `758a04a5a`
Governed wait	[code]	`runner.py`: `_wait_with_governance`, `_effective_max_runtime`, `_terminate_process_tree`
Limits not enforced elsewhere	[code]	`run_ledger.request_kill` (one caller, `runs_cli.py`); `_spend_enforcement`; `fleet_status.py`
Limit "never enforced", 5 h 6 min run, no kills	[notes]	agents' working notes, 2026-09-09
Per-child deadline versus one run row	[inferred]	`_run_child_agent`, `_run_start_parameters`
Proof of work, unplanned	[code]	`runner.py`: `_handle_success`, `_verify_work_complete`, `_detect_orphaned_commits`, `_push_and_verify_branch`
Proof of work, planned child	[code]	`root_dispatch_reconciliation.py`: `validate_leaf_result_commit`, `reconcile_successful_child`, `child_close_reason`, `FAILURE_CODES`

Claim or section	Tag	Anchor
Child outcome protocol	[code]	`child_outcomes.py`; `contracts/planned_root.py`: `ChildOutcomeKind`, `AcceptanceNotExecutedReason`; commits `e17b48214`, `de8ca8eaa`
Acceptance evidence and deferral	[code]	`acceptance_evidence.py`: `classify_child_acceptance`, `ROOT_UNVERIFIED_REASON_CODE`; `deferred_child_close_reason`; commit `344dd8b73`
Both switches default off	[code]	`RunnerConfig` in `runner.py`; `ops/yesod-birth/README.md`, `birth-runner.sh`, `credential-runner.sh`
H1 children closed untested	[notes]	agents' working notes, 2026-11-19
Execution placement	[code]	`execution_placement.py`, `execution_broker.py`, `execution_provider.py`; `docs/architecture/aws-lambda-microvm-execution-adapter.md`
Production placement is local	[deploy]	as of the adapter document (2026-08-25) and the stdin rollout runbook; no contrary evidence on `main`
Failure axes	[code]	`reconciliation.py`: `VERDICTS`; `failure_analysis.py`: `FAILURES`
Bounded retries	[code]	`runner.py`: `_handle_failure`, `_maybe_escalate_dispatch`, `INTERRUPTED_CHILD_MAX_LAUNCHES`

Not verified:

- The live value of `disposable_postgres_enabled` and `acceptance_hard_block_enabled` on each production runner on 2026-11-19. Source and birth scripts give `false`; the notes' account of H1 is consistent with that.
- Whether every production runner has the scoped agent database credential, or still uses the warning fallback.
- The explanation offered for a run exceeding its recorded limit (one run row, a fresh deadline per child) is inferred from two functions and was not traced end to end. The 5 h 6 min figure, the "none of thirteen killed" count and the phrase "never enforced" are from the working notes only; on `main` the owning runner's wait loop does enforce the limit per agent process.
- The H1 details (nineteen children, every close reason deferred, a failing test found afterwards) are from the working notes only.
- Whether any production run has used `lambda-microvm` placement or the per-run inference gateway lease. None was found.

- Whether the Codex standard-input transport has been rolled out to every live runner. The code and the rollout runbook are on `main`.
- What `note_status_writer.write_status` does with the requested status `blocked`, which is not a state in the current lifecycle definition (Chapter 14, Failure Is a State Machine, treats `blocked` as a disposition). The disposition write is verified; the status write was not traced.
- How `over_cap` is surfaced to operators in practice; only the labeling code comment in `fleet_status.py` was read.

8 The Merge Controller

This chapter explains how a finished branch becomes a commit on `main`. It defines the terms the merge controller is built from (lane, attempt, lease, fence), follows one change through the path that works, and then takes the deviations one at a time: a red gate, a stalled owner, a target that moved, a queue, a conflict, a closed door, a dead process. Two of them are told through incidents from October 2026 in which the controller kept `main` correct and stopped merging anyway.

Merging earns a chapter because parallel work is cheap to start and expensive to reconcile. Every change made in its own worktree eventually has to land on the same `main` and be true at the same time as all the others. The ways this goes wrong are ordinary:

- Two agents edit the same file. Each branch is fine alone and they conflict together.
- A branch is cut from `main` and `main` moves. The branch still merges cleanly, and it carries a test that pins behavior `main` has since changed.
- The gate passes and the process dies one step before the push. A second process has to answer "did it land?" from durable state.
- Two processes both believe they own the merge for the same repository.

Handled by guesswork, any of these corrupts shared history, which cannot be cheaply repaired. So in August 2026 Yesod replaced its first merge engine, a standing agent that polled a queue and launched a worker per batch, with a deterministic service. The **merge controller** (`yesod refinery controller`, the core of the subsystem the code calls the refinery) reads durable rows in PostgreSQL, derives one next action, takes it under a lease, and refuses with a named reason when the evidence is missing or contradictory. Coding agents never write `main`.

The controller is deterministic in the sense that it never guesses. It still uses a model: when two changes conflict, it sends the problem back into the factory as a bounded coding job and treats the result as an untrusted proposal that must pass the full gate again ("When the Merge Itself Conflicts").

8.1. The Vocabulary of a Merge

The design is small and leans on a handful of terms with exact meanings.

- **Refinery lane.** The unit of merge serialization: `LaneKey(repository, target_branch)` in `refinery_state.py`, stored as a row in `refinery_lanes`. "The `yesod` repository's `main`" is one refinery lane; another tool's `main` is another. This is a different thing from a dispatch lane, which names a model target (Chapter 16, Cost-Aware Dispatch). In this

chapter "lane" always means the refinery lane. The target is usually `main` and may be any other branch. The reviewed controller document on `main` configures six lanes.

- **Attempt.** One try at integrating one note into one lane: a row in `refinery_attempts` with a lifecycle. An attempt is immutable once it ends. A retry is a new row that points back through `prior_attempt_id` (`request_retry`).
- **Candidate.** The exact commit being considered: the source branch merged onto an exact base commit, identified by its SHA. The next section shows that it is a real, reproducible commit.
- **Lease.** A time-boxed claim on an attempt held by the controller process working it. `LeaseAuthority` carries an owner, an expiry and a fencing token.
- **Fencing token.** The lane's `fencing_generation`, a counter that only increases. Every claim and every reclaim increments it and stamps the new value on the attempt. A write that presents an old token is refused.
- **Merge turn.** `refinery_lanes.current_attempt_id`: the one attempt allowed to push this lane next.
- **Admissions pause.** A durable, global stop on intake, guarded by a revision number and a token. "When the Door Is Closed" covers it.

An attempt is in one of seventeen states, frozen in `AttemptState` and repeated in the database check `refinery_attempts_state_check`.

Kind	States
Active (7)	`requested`, `claimed`, `preparing`, `gating`, `gate_green`, `pushing`, `deploying`
Terminal, good (1)	`succeeded`
Terminal, other (9)	`gate_red`, `superseded`, `cancelled`, `timed_out`, `needs_rework`, `needs_operator`, `push_raced`, `infrastructure_failed`, `merged_not_deployed`

The edges between them are a strict forward map, `_ATTEMPT_TRANSITIONS`. A terminal state has no outgoing edge, and the trigger function `refinery_reject_terminal_attempt_mutation` rejects a write to a terminal row even from code that skipped the map.

8.2. The Good Case

The example is the change from Chapter 1 (Follow the Change): a read-only command that explains why an armed note is not running. A worker has committed it to a branch, the runner has pushed the branch, and the note is at `awaiting_merge`.

The runner enrolls the note by writing a row into `merge_queue`, the table the old engine used. The controller is now that table's only consumer. On its next pass, `reconcile_legacy_intake` creates an attempt in `requested` and marks the queue row `held`. On a queue row that word means "linked to an attempt and owned by the controller" (`_mark_queue_linked`). A note

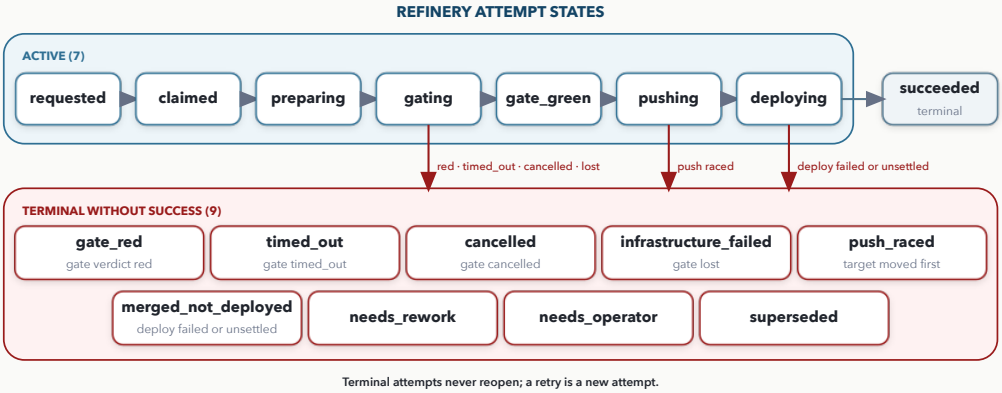

Figure 8.1.: The seven active and ten terminal states of a merge attempt, with the forward edges between them.

with a `held` disposition is a different thing: someone has told the factory not to deliver it yet (Chapter 5, Capture, Plan, Deliver).

From there the controller advances the attempt one bounded step per pass. Each step is one `ControllerAction` chosen by `LaneController` from the rows it can read, and each is authorized against the lease first:

> `requested` → `claimed` → `preparing` → `gating` → `gate_green` → `pushing` → `deploying` → `succeeded`

Claim. `claim_next_attempt_bounded` finds the attempt, increments the lane's `fencing_generation`, stamps the token and the lease on the attempt, and, when the claim owns the merge turn, sets `current_attempt_id`. Before it does, `evaluate_source_note_eligibility` re-reads the note under lock. If the note has been held or closed in the meantime, the attempt is cancelled with `note_no_longer_eligible`. That lazy cancel is the whole merge-side mechanism behind holding a delivery.

Prepare. `GitRepository.prepare` fetches the target and source branches, verifies that they are at the exact recorded SHAs, runs `git merge --no-commit --no-ff` in a worktree private to the attempt, writes the tree, and creates the candidate with `git commit-tree` and two parents: the base and the source. The commit message is fixed:

```
refinery: deterministic candidate

Target: main
Base: <base sha>
Source: <source sha>
```

Author and committer are a fixed refinery identity, and `_deterministic_identity_env` sets both dates to one second after the later of the two parents' commit times. Nothing in the commit depends on when or where it was built, so the same base and source always produce the same candidate SHA. A controller that crashes and prepares again gets the same commit

and can recognize it. The candidate is published under `refs/yesod/refinery-candidates/` (`candidate_ref_for`) so the gate can fetch it.

These commits fill the history of `main`. Of the 3,084 commits on `origin/main` between 2026-08-20 00:00 UTC and this edition's snapshot (`16606a42`, commit date 2026-11-17), 410 carry the subject `refinery: deterministic candidate`, and each is a merge the controller landed. Two consequences catch people. The commit date is synthetic, so `git log` dates do not say when a merge happened: the agents' working notes for 2026-10-13 record a report of "no merge for 5.5 h" taken from commit dates when the real gap was about 1.3 hours. And the subject never mentions the note, so the reliable test that a branch landed is `git merge-base --is-ancestor`, or the attempt row.

Gate. The controller stages the candidate, submits it to the lane's gate, and reconciles the gate execution on each pass until it is terminal. The gate is Chapter 9 (The Deterministic Gate). A green result moves the attempt to `gate_green`.

Push. `push_ref` pushes the candidate SHA to the target branch with `--force-with-lease=<ref>:<base>`. That is a compare-and-swap: the remote accepts the push only if the branch still points at the base the candidate was built on. The push is recorded as a durable external effect (`target_push_effect_key`) before it is made, so a process that dies around it can find out afterward whether it applied.

Deploy. A lane without a deployment block goes from `pushing` straight to `succeeded`. The `yesod` lane has one, so its attempt enters `deploying` with reason `push_confirmed`. The controller runs the lane's deployment command and an independent observer, and only an observer reporting the exact candidate SHA counts as applied. If that arrives, the attempt ends `succeeded`. If the observer fails, or stays pending past the settle budget, the attempt ends `merged_not_deployed` and the lane is freed: the commit is on `main` and is not running. According to the agents' working notes, through most of the autumn that was the usual ending on the `yesod` lane, for reasons that belong to Chapter 10 (Merged Is Not Deployed), which owns this state and what it does to the note.

That is the whole path: one lease, one candidate, one gate, one push, with no batch and no global lock. The rest of the chapter is the failure cases.

8.3. When the Gate Says No

The most common failure is a red gate. The controller ends the attempt at `gate_red`, and `_project_rework` marks the note for rework with the failure evidence attached. Nothing retries on its own. When the branch has been fixed, `yesod refinery rearm` (a mutating command) enrolls it again, and that creates a new attempt.

On 2026-11-16 the fix note that followed the mid-November controller-replacement attempts entered a merge window and ran `requested` → `claimed` → `preparing` → `gating` → `gate_red`, on one test that leaked a process umask into its test worker (agents' working notes, 2026-11-16). That attempt is terminal and stays `gate_red`. The note merged later on a different attempt, as the candidate commit `16606a42` (stamped 2026-11-17 19:16Z, a time derived from its source commit, not the time of landing); the working notes place the merge window on 2026-11-18.

> **Invariant — A terminal attempt never becomes active again.** `allowed_attempt_transitions` returns an empty set for every terminal state, and `refinery_reject_terminal_attempt_mutation` enforces the same rule in the database. A late or crashed writer cannot put a dead merge back into `gating`, because that edge does not exist.

Immutability also keeps the accounting honest: a note that merges on its third attempt cost three gate runs, and all three rows survive. The cost is that a red note waits for someone to act. The working notes for 2026-09-09 record three notes sitting in `awaiting_merge` behind terminal `gate_red` attempts, one of them for about 23 hours, because nothing was responsible for noticing.

8.4. When the Owner Stalls

The controller claims an attempt, starts gating, and its host freezes. From outside, frozen and dead look the same.

The lease has an expiry. When it lapses, the scheduler's next pass calls `reclaim_expired_attempt`, which increments the lane's `fencing_generation` and takes the higher token. If the original owner wakes and tries to write, `authorize_attempt_transition` checks its lease before it even looks at the requested state edge and raises `LeaseAuthorizationError`. The function's docstring gives the reason for the order: a stale worker should learn nothing that could tempt it to continue.

> **Design Trade-off — Fencing over coordination.** Yesod does not try to make two owners agree on who is in charge. It makes the stale owner's writes impossible to commit. The price is that a lane whose owner has died waits for the lease to run out, and the reviewed controller document on `main` sets `lease_seconds` to 30600 (eight and a half hours).

A small production proof exists. On 2026-09-10 an operator restarted the controller nine seconds after a gate finished, while the attempt was still `gating` under a lease owned by the killed process. Nobody intervened. The lease expired, the new process reclaimed with a higher token, and the attempt continued about eleven minutes later with no lost work (agents' working notes, 2026-09-10). The test for a safe restart is therefore "no held lease", which is stricter than "no running gate".

8.4.1. The Second Fence

The fencing token protects an attempt from a stale process. It says nothing about a stale controller host. In October the plan for deploying the controller changed from updating it in place to replacing its VM (Chapter 11, Deploy Is a Host Swap), and a replacement needs a moment after which the old controller can write no lane. The controller-leadership fence was added for that on 2026-10-13 (`485a9129c`, migration `0118`). The table `refinery_controller_lane_leadership` holds, per lane, a `holder_id`, a `leadership_generation` and a lease expiry. Every pass of the controller begins by renewing its leadership and proving it with `_require_controller_leaderships`. Claims, intake and admission reconciliation

all carry that proof, and a mismatch raises `ControllerLeadershipConflict` before the first lane mutation. Attempts record the holder and generation that wrote them.

On a single serving controller this fence is always satisfied. Its purpose is the handover, which has never run in production as of 2026-11-19. Chapter 11 says what has and has not been proven.

8.5. When `main` Moved Underneath

The candidate was built on a specific base. If the target branch advances between gating and pushing, the compare-and-swap push is rejected, and the attempt ends `push_raced` with reason `target_advanced_before_push`. The controller never forces a stale candidate over newer history. It queues a retry itself (`ENSURE_RACE_RETRY`), and the new attempt builds and gates a fresh candidate on the new base. The pushed commit is therefore always the gated commit.

Building every candidate on current `main` is also what exposes a branch that merges cleanly and still fails the gate because `main` has moved; Chapter 9 calls this semantic staleness.

8.6. When Another Change Wants the Same Branch

Two notes target `yesod/main`. They are the same lane, so they serialize: one attempt holds the merge turn and the other waits. The claim predicate in `_claim_next_admissible_attempt` enforces it, backed by the unique indexes `uq_refinery_attempts_active_lane` and `uq_refinery_attempts_a` A note on a different lane merges concurrently. No batch is formed, so none needs bisecting when something goes red.

The August edition of this book said head-of-line blocking was avoided structurally. That is true across lanes and false inside one. Inside a lane the order is strict first-in, first-out by `attempt_number`: the claim query sorts by `requested_priority`, and then refuses any attempt while a lower-numbered attempt in the same lane is still `requested`. Priority orders work across lanes, and a re-enqueue at a higher priority supersedes the same note's own waiting attempt (reason `priority_reenqueue_superseded`). It does not move a note ahead of other notes. The working notes record this being learned twice: on 2026-10-12 a priority-0 re-enqueue was announced as gating next with nine attempts ahead of it, and the mistake recurred on 10-14. The only way to reorder a lane is to hold the notes in front.

One overlap is allowed. Since 2026-10-10 a lane can let the next attempt begin gating while its predecessor is in `deploying` (`admit_during_settle`). That saved time and produced this incident.

> **Factory Floor — Merge-turn theft, 2026-10-12.** Twice in one day, at about 10:20Z and 23:15Z, the `yesod` lane stopped with a `gate_green` attempt that never pushed, and a `UniqueViolation` on `uq_refinery_attempts_active_lane_pool` on every pass. Attempt N had been claimed while its predecessor was `deploying`, so N was gating without owning the merge turn. When the predecessor settled, the turn was cleared, and the next claim gave it to attempt N+1. N was green and could not push because the turn was not its own. N+1 held the turn and could not reserve a gate pool because N still had it. Each occurrence cost 20 to 30 minutes and was cleared by hand with a guarded update pointing `current_attempt_id` at the green attempt (agents' working notes, 2026-10-12). The fix, `779673ec0` on 2026-10-13, makes a claim own the merge turn only when no older attempt in the lane is still non-terminal. `main` was never wrong during either stall. The general lesson: a released resource needs a rule for who receives it, in order of precedence.

The base of the next attempt needed a rule too, because a successor cut from `main` while its predecessor is green and unpushed is certain to end `push_raced`. Since 2026-10-13 an ordinary successor waits for its predecessor's `push_confirmed` and builds on that pushed candidate; only an attempt explicitly stacked on the predecessor's candidate may start earlier. The guard that enforces "build only on a target the controller can prove was pushed" caused the second incident one day later.

> **Factory Floor — The guard that read history, 2026-10-14.** Minutes after the controller was promoted to a release containing the new guard, the first attempt on the `yesod` lane sat in `preparing` and never cut a candidate. The guard compared the base with the lane's `last_observed_target_sha`, and that column advanced only when an attempt reached `succeeded`. Under the operating policy of the time, 51 consecutive merges had ended `merged_not_deployed`, so the column still held a SHA from 2026-10-09. The lane waited for a deployment, the deployment waited for merges, and the merges waited for the lane (agents' working notes, 2026-10-14). Every pre-run of the change had passed, because each started from an empty database with no lane history. Commits `fb90ae96e` and `62b5842b9` landed the same day, and on `main` `resolve_lane_pushed_target` takes the most recent attempt in `deploying`, `succeeded` or `merged_not_deployed`. That these commits were the response is inferred from their date and subject. The general lesson: before promoting controller logic, list the durable state it reads and look up its value in production, because tests that start from empty cannot see it.

8.7. When the Merge Itself Conflicts

Serialization orders two changes without making them compatible. When `git merge` reports a conflict during `preparing`, the attempt ends `needs_rework` with reason code `merge_conflict`. The controller does not attempt a resolution.

On a later pass, `reconcile_conflict_repairs` creates a row in `refinery_conflict_repairs` for that attempt, with its own small lifecycle: `requested` → `armed` → `running` → `awaiting_gate`, ending `succeeded`, `failed` or `cancelled`. At most one repair per note is active (`uq_refinery_conflict_repairs_active_note`), and the count per note is capped

by `max_repairs_per_note`, which defaults to 2 in `ConflictRepairPolicy` and is 1 in the reviewed controller document.

Arming removes the failed queue row, returns the note to `open`, and chooses a model target by a deterministic balance policy (`RepairLanePolicy.decide`). One credential-free read of a dashboard snapshot decides between a primary and a fallback dispatch target; if neither is usable the repair parks with `repair_lane_credits_exhausted` or `repair_lane_unavailable`. The code defaults are `codex-astra-xhigh` and `claude-fable-5`. The reviewed document names a different primary, and which model runs repairs in production today was not verified for this edition.

The coding agent that picks up the repair works on the note's original source branch and is granted exactly one command that workers are otherwise forbidden: `git merge --no-ff --no-edit origin/<target>`. It resolves the conflict and runs targeted tests. The runner, not the agent, then pushes the source branch with `--force-with-lease`. The new source SHA then enters as a fresh attempt and runs the whole path from `requested`, full gate included. The repair row reaches `succeeded` only when that attempt does, never on the agent's exit code. When the budget is exhausted, the controller opens an operator-inbox item (`conflict_repair_budget_exhausted`) asking a human to repair the branch or supersede the note.

The model proposes a resolution and the gate decides whether it becomes history. That division has held. Getting a repair picked up has been less reliable. In September, repairs were armed while runner slots sat idle, because a note whose planned children were all closed offered the runner nothing to claim; `7c8ff9b41` (2026-09-11) made the repair row itself the claim authority. Four other repairs that week had been armed correctly and were then pulled back into planning by a recovery procedure, where an armed repair cannot be claimed (agents' working notes, 2026-09-11).

8.8. Before the Gate

A lane is serial, so a candidate that was always going to fail the gate costs every note behind it. The pre-gate, built on 2026-10-11 and 10-12, is a cheaper check between `preparing` and `gating`: read-only merge-tree conflict checks against `main` and against attempts queued ahead, then a deterministically selected subset of tests run first on `main` and then on the candidate, counting only new failures.

A lane's `pre_gate_mode` is one of `PreGateMode`: `off`, `shadow` or `active`. In `shadow` a red verdict is stored and the candidate goes to the real gate anyway. In `active` a failed pre-gate ends the attempt for rework.

> **Proven Where — as of 2026-11-19.** The pre-gate is in code on `main`. The default mode is `off`, the reviewed controller document on `main` sets no `pre_gate_mode` on any lane, and the activation runbook carries the status line "procedure only; qualification and activation have not been performed". The working notes indicate that an early form ran against live attempts in mid-October, and record that on 2026-10-13 a pre-gate key that was not scoped to its attempt wedged the lane head for about 30 minutes (fixed by `040207f32` on 2026-10-29). The pre-gate is not enforcing anything according to `main`; the live mode was not captured for this edition.

8.9. When the Door Is Closed

Changing the controller's configuration, onboarding a repository, promoting a release and replacing the controller host all need a period in which no new merge begins.

The mechanism is one row. `refinery_admission_control` has a single `global` row with a `paused` flag, a `revision` and a `pause_token`. `pause_admissions` locks the row, checks that the caller's `expected_revision` equals the current one, sets `paused`, increments the revision and generates the token. `resume_admissions` requires the current revision and the exact token of the active pause, and raises `AdmissionConflict` on either mismatch. Both append to `refinery_admission_events` with the actor, the reason and a SHA-256 of the token.

Admission and pause linearize because every admitting transaction calls `require_admissions_open`, which reads the same row `FOR SHARE` and raises `AdmissionPaused`. An enrollment either commits before the pause and is visible to whoever paused, or sees the pause and is refused.

Since a change of 2026-11-09 (`3c7ab139c`) the token is never printed. `yesod refinery admissions pause` writes the token and its revision to a new private file named by `--token-out`; `resume` reads it back with `--pause-token-file`; `status` shows only an eight-character fingerprint (`admission_token_fingerprint`). Both mutating verbs also require `--expect-revision`, `--actor`, `--reason` and a literal confirmation word.

What the pause covers is narrower than its name suggests. `require_admissions_open` has four call sites on `main`:

Call site	What is refused
`_enqueue_impl`	a new `merge_queue` row
`_reconcile_impl`	queue reconciliation
`reconcile_legacy_intake`	turning queue rows into attempts
`refinery_branch_submission`	`yesod refinery submit-branch`

Claiming, gating and pushing do not check it, so an attempt admitted before the pause runs to its end while intake stays closed. The pause does not stop runners or planning. And the token proves only that the resumer holds the pause; the row records whose pause it is only as the `actor` string the caller supplied. Procedures that need to know the lanes are quiet, or that a particular person closed the door, build that proof on top (Chapter 11; Chapter 18, Signed Human Authority).

In current practice admissions are paused most of the time and merging happens in windows: resume, let the intended note's attempt come into being, pause again. Two properties of the mechanism show in the record. Intake is fast: in the window that merged `16606a42` (placed by the notes on 2026-11-18), reconciliation enrolled the waiting note 1.8 seconds after the resume. And an open door admits whatever is eligible: on 2026-11-09 a migration note that was meant to wait finished building inside another note's window, and intake created an attempt for it 20 seconds before the hold poller reached it. It was cancelled only because the lane was busy (agents' working notes for both dates).

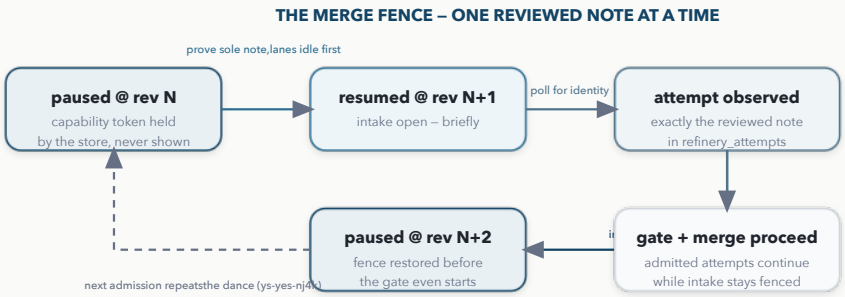

Figure 8.2.: A merge window uses the admissions pause three times: resume at one revision, observe exactly the intended attempt, and pause again at a new revision while that attempt continues.

8.10. When the Process Simply Dies

The controller can die between any two steps. A pass keeps no authoritative state in memory: the attempt's state, the lease, the fencing token, the candidate SHA and the record of each external effect are rows, and by the rule stated at the top of `refinery_controller.py` no PostgreSQL transaction spans Git, provider or deployment I/O. The next process re-derives its position from the rows, and where an external call's outcome is unknown it reconciles against the outside world before acting again. A crashed pass costs one pass.

8.11. The Tick, Assembled

One pass of `DeterministicRefineryService.run_once` is called a tick. The reviewed document sets the interval between ticks to five seconds. In order, a tick:

1. checks that this controller is active, renews its lane leadership and proves it;
2. runs post-merge settlement, which closes notes whose merged candidates have since been observed deployed (Chapter 10);
3. runs conflict repair before intake, so that a conflict that ended on a previous tick is removed from the queue before intake can enroll it again;
4. bridges `merge_queue` into `requested` attempts, unless admissions are paused;
5. re-checks the source note of every attempt not yet submitted to a gate (`reconcile_attempt_admissi`
6. runs the scheduler: renew the leases it owns, reclaim expired ones under a higher token, claim new attempts up to `max_active_lanes`, and advance each owned attempt by one action;
7. runs conflict repair again, so a conflict recorded in this tick is armed at once;
8. evaluates alarms;
9. lists outstanding recovery work and routes anything ambiguous to the operator inbox.

The alarms are advisory and are delivered through Yesod mail. There are three families:

repair alarms (warning after `WARNING_AFTER_SECONDS = 900`), progress alarms for lane heads and planning jobs (per-state floors such as `REQUESTED_FLOOR_SECONDS = 600`), and dispatch alarms for runner starvation. Exceptions from the progress and dispatch alarms are caught in `run_once`, so they cannot fail the tick.

One check is not advisory. On 2026-09-09 a gate execution was created whose provider job never launched. The attempt never ended, the lane stayed occupied, and for four hours the controller reported itself fresh with a tick error count of zero while merging nothing (agents' working notes, 2026-09-09). The stall sentinel added the next day (`d26f7eab6`) is `_requested_without_execution_stall_error`: if any configured lane has an attempt that has been `requested` for `STALL_GRACE_MINUTES = 15`, no gate execution is live, and admissions are not paused, the tick itself reports the error `RequestedWithoutExecutionStall`. A companion commit the same day (`5a465dd9c`) added `reconcile_terminal_lane_slots`, which releases a merge turn still pointing at a terminal attempt.

When a lane is not moving, the read-only commands `yesod refinery status` and `yesod lane show` answer the first three questions: which attempt holds `current_attempt_id`, what state it is in, and whether admissions are paused.

Every step is bounded and safe to repeat. No incident in this chapter put a wrong commit on `main`. Several stopped a lane for minutes or hours, and the red gate that nobody followed up left a note waiting for most of a day.

The controller decides whether a candidate may join shared history and when. Whether the candidate is correct is decided by the gate, which runs the tool's own tests against the exact candidate tree and is the subject of the next chapter.

Source Notes

Claim or section	Tag	Anchor
Opening; controller role	[code]	`yesod/docs/REFINERY.md`; `refinery_runtime.py` `DeterministicRefineryService`
Vocabulary; states and edges	[code]	`refinery_state.py`: `LaneKey`, `AttemptState`, `_ATTEMPT_TRANSITIONS`, `LeaseAuthority`
Six lanes; poll 5 s; `max_active_lanes` 2; `lease_seconds` 30600; repair cap 1	[deploy]	`ops/yesod-dispatch/config/controller-reviewed-document.json` on `main` (reviewed mirror, not a live capture)
Intake bridge; queue `held`	[code]	`refinery_intake.py`: `reconcile_legacy_intake`, `_mark_queue_linked`
Claim; eligibility	[code]	`refinery_control.py`: `claim_next_attempt_bounded`, `_claim_next_admissible_attempt`; `refinery_note_eligibility.py`

Claim or section	Tag	Anchor
Deterministic candidate; CAS push	[code]	`refinery_git.py`: `GitRepository.prepare`, `_deterministic_identity_env`, `candidate_ref_for`, `push_ref`
410 of 3,084 commits	[code]	`git rev-list --count --since=2026-08-20T00:00:00Z origin/main` at `16606a42`
Synthetic-date misreport	[notes]	agents' working notes, 2026-10-13
`deploying`, `merged_not_deployed`	[code]	`refinery_controller.py`: `_run_deployment`, `_merged_not_deployed`
Red gate; rearm	[code]	`refinery_intake.py` `_project_rework`; `refinery_cli.py` `rearm`
2026-11-16 red attempt; 23-hour wait	[notes]	agents' working notes, 2026-11-16 and 2026-09-09
Lease, reclaim, fence order	[code]	`refinery_state.py` `authorize_attempt_transition`; `refinery_control.py` `reclaim_expired_attempt`, `renew_lease`
2026-09-10 restart	[notes]	agents' working notes, 2026-09-10
Leadership fence	[code]	migration `0118`; `refinery_control.py` `_require_controller_leaderships`; commit `485a9129c`
Push race	[code]	`refinery_controller.py`: `SETTLE_PUSH_RACED`, `ENSURE_RACE_RETRY`, reason `target_advanced_before_push`
FIFO; priority; overlap	[code]	`_claim_next_admissible_attempt`; `_supersede_lower_priority_requested_attempts`; indexes from migrations `0041`, `0078`, `0079`; commit `6c8c5444f`
Priority misread twice	[notes]	agents' working notes, 2026-10-12 and 10-14
Merge-turn theft	[notes] + [code]	agents' working notes, 2026-10-12; commits `779673ec0`, `8534c1c89`, `67326a577`
History-dependent guard	[notes] + [code]	agents' working notes, 2026-10-14; `refinery_control.py` `resolve_lane_pushed_target`; commits `6703cfed0`, `fb90ae96e`, `62b5842b9`; causal link [inferred]
Conflict repair	[code]	`docs/refinery-ai-conflict-repair.md`; `refinery_conflict_repair.py` `ConflictRepairPolicy`; `refinery_repair_lanes.py` `RepairLanePolicy`; commit `7c8ff9b41`
Stranded repairs	[notes]	agents' working notes, 2026-09-11

Claim or section	Tag	Anchor
Pre-gate	[code] / [deploy]	`refinery_pre_gate.py` `PreGateMode`; `RuntimeLaneSettings.pre_gate_mode`; `docs/runbooks/yesod-pre-gate-shadow-activation.md`; commit `040207f32`
Admissions pause	[code]	`refinery_admission.py`: `require_admissions_open`; `refinery_cli.py`; commits `eb2f93d23`, `3c7ab139c`
Window timings (1.8 s; 20 s)	[notes]	agents' working notes, 2026-11-18 and 2026-11-09
The tick; alarms; stall sentinel	[code]	`refinery_runtime.py` `run_once`; `refinery_alarm.py`; `STALL_GRACE_MINUTES`; commits `d26f7eab6`, `5a465dd9c`
Four-hour stall	[notes]	agents' working notes, 2026-09-09

Not verified:

- Live values of the controller settings and of each lane's `pre_gate_mode`. The numbers printed are from the reviewed document on `main`.
- Which model target runs conflict repairs in production today, and the effect of `c6d5fd8bd` (2026-10-27) on the targets chosen at arm time.
- That `fb90ae96e` and `62b5842b9` were written in response to the 2026-10-14 deadlock. The commits have no bodies; the link is by date, tracker id and the behavior of `resolve_lane_pushed_target` on `main`.
- How a restarted controller recovers its own unexpired leases under the current 30600-second lease. The 2026-09-10 recovery in about eleven minutes predates that setting (the reviewed document has carried that value since it was added on 2026-09-18, `a1eeff8d9`).
- When `16606a42` actually landed. Its commit timestamp is derived from its source commit (one second later), and the working notes place the merge window on 2026-11-18.
- That an early form of the pre-gate ran against live attempts in mid-October; this rests on one working note of 2026-10-13.
- All incident durations and counts (20 to 30 minutes, 51 merges, nine attempts ahead, 23 hours, four hours, eleven minutes, 1.8 seconds, 20 seconds) come from the agents' working notes and were not measured from the database for this edition.

9 The Deterministic Gate

The merge controller (Chapter 8, The Merge Controller) decides whether a candidate may join shared history and when. It never decides whether the candidate is correct. That judgment is delegated, every time, to the **gate**: the authority that tests one exact candidate tree and returns a verdict.

This chapter explains the contract around that verdict and the machinery that produces it for Yesod's own repository. The contract is the same one the August edition described: test the exact candidate, report in provider-neutral states, keep "the tests failed" separate from "the tests could not run", and never fall back silently to a weaker environment. The machinery changed completely. In August the Yesod suite ran on disposable cloud MicroVMs. Since early September it runs on nine disposable virtual machines on the home hypervisor `sefer`, as one concurrent run called a union. The chapter says why the gate came home, follows one union from the candidate commit to nine receipts and a coverage proof, and then covers the two ways a gate deviates: a red verdict, and no verdict at all. It ends with what gate-scaling work is live and what is only staged on `main`.

A gate is harder than "run pytest and read the exit code". A false green is unrecoverable, because a broken commit becomes `main` and every later change builds on it. The environment has to be a constant, or the verdict describes the machine instead of the code. A test run can also hang or vanish, so "did not pass" and "never answered" must be different recorded facts.

9.1. The Vocabulary of a Gate

- **Gate.** The merge-time test gate for one refinery lane (one repository and branch). "Gate" alone always means this. The approval steps that the dispatcher places on beads are a different mechanism and are called dispatcher approval-gates.
- **Candidate SHA.** The exact commit the controller built in `preparing` by merging the delivered branch onto current `main`. The gate tests that commit and nothing else, and on green the controller pushes that same commit.
- **Verdict.** A statement that the candidate was evaluated and judged: green or red.
- **Infrastructure failure.** The gate could not run or could not report. No judgment about the candidate exists.
- **Gate execution.** One concrete run on one candidate, tracked by `GateExecutionState` in `refinery_state.py`: `submitted` → `starting` → `running`, then one of `green`, `red`, `timed_out`, `cancelled`, `lost`.
- **Provider.** Where the gate runs. `GateProviderName` in `refinery_gate_provider.py` has four members: `local`, `microvm`, `sefer`, `container`.

- **Broker.** The controller-side half, `GateBroker` in `refinery_gate_broker.py`. It writes the intent to gate into PostgreSQL before any external call, then calls the provider outside any database transaction. Its module docstring calls the path "deliberately synchronous and memoryless".
- **Idempotency key.** A deterministic fingerprint of one gate on one candidate (`gate_execution_idempotency_key`). A controller that crashed mid-gate asks the provider what became of that key instead of launching a second run.
- **Demon.** One gate VM on `sefer`, named `yesod-gate-dg<N>-<demon>-<suffix>` after a demon of Jewish folklore (samael, dybbuk, lilith). The number is the gate fleet generation. It labels the birth recipe the VM was built from and does not count replacements (`ops/yesod-birth/README.md`: "Generation is a recipe label").
- **Union.** One concurrent gate run on nine demons: eight shards and one safety island. The provider verdict carries `provider_shape: "sefer_parallel_union"`.
- **Shard and safety island.** A **shard** runs one slice of the suite with four parallel pytest workers. The **safety island** runs, with no parallel workers, the few tests known to be unsafe under parallel execution.
- **Role.** The stable identity of a slot in the union: `sefer-shard-0` through `sefer-shard-7`, and `sefer-safety-island`. Host names change when a demon is reborn; roles do not.
- **Carrier.** The Git bundle that transports the candidate to each demon.
- **Receipt.** The durable record that one step happened for one role.

The distinction between a verdict and an infrastructure failure carries most of the chapter. Nearly every incident below is a case where one was mistaken for the other.

9.2. Four Providers

All four providers implement one protocol, `GateProvider`, with five operations: `submit`, `find_by_idempotency_key`, `observe`, `cancel`, `fetch_artifacts`. The controller sees only `GateExecutionState`, whichever provider ran the tests.

Provider	Used by	Where it runs
`sefer`	the Yesod repository	nine demons on `sefer`
`local`	small tool lanes	a subprocess on the controller
`container`	the `sjbis` Rust lane	a rootless container host
`microvm`	no lane	dormant code

The assignment comes from `docs/refinery-fleet-gates.md` and from the reviewed controller document on `main` (`ops/yesod-dispatch/config/controller-reviewed-document.json`, last changed 2026-10-12), which gives the Yesod lane `gate_provider: "sefer"` and `gate_profile: "sefer-parallel-py3146-8x4-v1"`. The small tool lanes stay local because, in the words of `docs/refinery-fleet-gates.md`, their suites "already complete in roughly 0.1–93 seconds" and remote startup would add more operational surface than throughput. The container provider landed on 2026-09-22 and 09-23 (`19f764260`, `5a35c087d`), and the

one Rust lane was put on it on 2026-09-24 (`f461ae0f9`). The MicroVM provider is still in the tree, `ops/yesod-dispatch/README.md` calls its poller "legacy", and no lane in the reviewed document selects it. The fleet cap and leak tripwire of the August edition belong to that dormant path.

9.3. Why the Gate Came Home

The move began with a disagreement between two environments. On 2026-08-28 a work note (`ys-yes-wmuw`, recorded in `docs/tools/yesod.md`) was filed whose text opens "Bind the refinery merge gate to exact CPython 3.14.6". Its context paragraph says that a candidate "passed all three explicit pristine CPython-3.14.6 gates but went gate_red in the authoritative refinery MicroVM gate". The gate's verdict was partly a statement about its own image. The same day two commits landed: `f56a845ac`, "add isolated Sefer fleet provider", and `1964730c3`, "make Sefer the pinned primary gate runtime".

The note's full reasoning is truncated in the repository copy, so the complete motive is not recorded there. Cost and dependence on a cloud account may also have mattered; that is inferred. The source does show the design response. The Sefer gate accepts exactly one interpreter invocation, a fixed absolute path to a pinned CPython 3.14.6 followed by `-I -m pytest` (`CANONICAL_PYTEST_COMMAND` in `refinery_sefer_gate.py`). The rollout runbook (`docs/runbooks/sefer-refinery-gate-rollout.md`) states that a bare `python`, a bare `pytest`, `uv run`, another Python patch release, or a command supplied by the candidate is "infrastructure-red". A lane on this provider may not even supply pytest arguments in its configuration (`LaneControllerConfig` in `refinery_controller.py` rejects `pytest_args` for Sefer lanes); they come only from an immutable partition manifest.

The same runbook records the last full-suite run on a single machine (9,343 passed, 50 skipped) and says it "must never be replayed". On 2026-09-01 and 09-02 the eight-by-four partition manifest and the parallel union landed (`64ebca833`, `2e77ba6b9`), followed on 09-04 by the candidate stager and isolated collection (`46ec1d209`, `9f1ad55b8`). The exact day the Sefer provider became the authority for the Yesod lane falls between 09-02 and 09-04 by commit evidence; no activation receipt is in the repository. On 2026-09-16 the controller document was switched to the first recipe-born fleet only (`2584d2844`), and the nine original hand-built gate VMs were shut down at 20:46:14Z and kept cold (`docs/runbooks/g1-cold-recovery-2026-09-16.md`).

9.4. The Good Case: One Union

Follow the example change from Chapter 1 (Follow the Change). Its branch was delivered, the controller claimed the Yesod lane, merged the branch onto current `main`, and built one candidate commit. The attempt is now in `gating`.

Reserve. Before any network call, `reserve_candidate_capacity` writes nine rows to `refinery_gate_reservations` in state `reserved`, one per role, against one gate pool. The rollout runbook's wording is "Partial admission is forbidden": a candidate gets all nine bind-

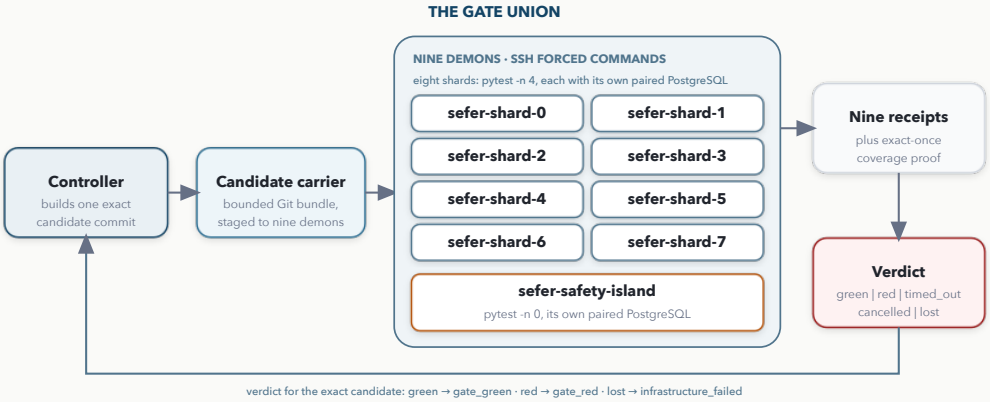

Figure 9.1.: One gate union: the candidate carrier staged to nine demons, one collection, eight shards and the safety island, nine receipts and the coverage proof behind a single verdict.

ings or none. A pool runs one gate at a time (`lock_pool_for_gate_execution` in `refinery_pool_state.py`).

Stage. The controller records the candidate as a bounded, content-addressed Git bundle (`record_candidate_bundle`) and writes one staging intent per role (`ensure_staging_intents`, table `refinery_gate_staging`). On each demon a stager unpacks the bundle and publishes a read-only checkout at a path keyed by the candidate SHA. Publication uses `renameat2` with `RENAME_NOREPLACE`, so an existing checkout is never replaced underneath a running job. The preparation phase (`GatePreparationPhase`) is `staging`.

Collect. One collection intent (`ensure_collection_intent`, table `refinery_gate_collections`) asks one demon to run pytest collection against the published checkout and return the sorted list of test node ids with its digest. Collection happens once, before any shard starts, so all nine roles work from one agreed list. The phase moves to `collecting`, then `prepared`.

Partition and prove. `record_candidate_parallel_plan` builds a `PartitionPlan`: eight lists of node ids, plus the island's list, which is whatever `SAFETY_ISLAND_SELECTORS` matches (one reservation test and the two release-promotion test modules). The active profile assigns each remaining node to a shard by a hash of its id (`sha256-domain-node-id-modulo-v1`). The plan carries a `CoverageProof`. Its builder, `_coverage_proof_record` in `refinery_gate_partition.py`, raises if any node is assigned twice or if the union of the nine lists differs from the collected list: "partition union does not cover every collected node ID exactly once".

Submit. The broker persists the submission intent, then `SeferGateProvider.submit` sends nine `submit` requests over SSH. The job id is derived from the idempotency key (`sefer_job_id`, prefix `sefer-v2-`), so, as the module docstring puts it, "a response lost after launch never causes a second shard process". Reservations move to `running`; the gate execution moves `submitted` → `starting` → `running`.

Run. Each demon runs its list in its own process group against its own PostgreSQL, a dispos-

able database container paired with that demon by name. Shards use `pytest -n 4`; the island uses `-n 0`. Progress and a per-test completion stream come back through `lookup` requests.

Settle. When all nine roles are terminal, the provider's `_observation` reduces them to one state: green only if every record passed. The controller fetches each role's artifacts (the pytest report log, status, standard output and error, and a runtime provenance record) into blob storage. The runbook states the full condition for green: "eight unique shard receipts, the genuine island receipt, exact-once coverage proof, matching release/runtime/profile/topology, distinct database bindings, complete artifacts". The gate execution becomes `green`, the attempt becomes `gate_green`, and the reservations are `released`. The controller then pushes the exact candidate SHA by compare-and-swap, as Chapter 8 describes.

A union measured on 2026-09-07 and recorded in `ops/yesod-birth/gate-profiles.json` ran two shards of 1,290 and 1,327 tests in about eleven minutes each, and the island's 52 tests in 7 seconds. A month later the commit that raised a controller deadline (`cb4e90efc`, approved 2026-10-06) put the whole gate at 44 to 54 minutes. The two figures are not measured the same way (one is a shard's run time, the other the whole gate), so the roughly fourfold growth is approximate. The repository does not record its cause. Growth of the test suite is the likely one, and that is inferred.

9.5. The Demon and Its Boundary

A demon is built to be thrown away, and Chapter 12 (Born, Not Repaired) describes how one is born from a template. What matters here is how little the controller can do to one.

The controller never gets a shell. Four locked accounts exist on each demon (`ops/sefer-gate/sshd_config.d/`): `gate-audit` for read-only inventory, `gate-job` for running tests, `gate-stager` for publishing the candidate, and `gate-credential` for delivering one database credential. Each accepts only public-key authentication from its own key file, permits no terminal, no forwarding and no tunnel, and has an SSH `ForceCommand`, so whatever the client asks for, the server runs one reviewed program. The job program, `yesod-sefer-gate-forced-command`, accepts seven actions (`submit`, `lookup`, `cancel`, `artifacts`, `preflight`, `collect`, `collect-lookup`). Its docstring states the rest of the boundary: request bodies "arrive as bounded JSON on stdin; they are never interpolated into a shell", and test subprocesses "receive a newly constructed environment" and always begin with the pinned interpreter prefix.

The SHA-256 of that program is part of the lane's pinned runtime identity in the live controller document (`provider_code_digest`, per `docs/runbooks/sefer-forced-command-rollout.md`). The worker hashes its own file and refuses a request whose pinned digest differs ("forced-command worker digest mismatch"), so changing the worker is a planned window that reinstalls every gate host and re-pins the document together.

> **Invariant — A demon keeps no secret across a reboot.** The only credential a demon ever holds is the one for its own disposable test database, and it holds it on tmpfs under `/run`, which is empty at every boot. The encrypted catalog on the controller (`sefer_gate_credentials.py`) is authoritative. `SeferCredentialSynchronizer` in `refinery_sefer_credentials.py` reads a names-only status through the `gate-credential` account, compares the demon's boot id with the one it last delivered to, and sends the credential envelope only when the status is stale or unready. The rollout runbook adds that controller, workers and gate subprocesses "receive no AWS credential variables".

That design was written after its absence was felt. The agents' working notes for 2026-09-09 record that gate database credentials held on tmpfs were destroyed by a reboot, with nothing to recreate them, and that PostgreSQL was down on eight of nine gate databases while the service manager reported the unit active. The catalog and boot reconciliation landed the next day (`a69954af0`, "persist Sefer gate credentials"). The boot unit that prepares the empty tmpfs layout had a trap of its own, which Chapter 12 tells.

Before a fleet is trusted, a fleet preflight (`ops/yesod-dispatch/scripts/preflight-sefer-gate-fleet.py`) proves a generated inventory against the topology: host key, machine identity, release, exact Python, manifest, storage, firewall policy, and the pairing of each role with its own database. Before each admission the controller runs a smaller per-pool probe, `SeferPoolReadinessProbe`. Both sides of that wire share one contract module (`sefer_preflight_contract.py`, `ffaadbfe0`), because two copies of a rule can pass their own tests and still contradict each other. A duplicated rule had already caused an outage on 2026-09-09: two copies of the gate hostname pattern both rejected the new fleet's naming, the rejection happened at controller startup, and pointing the topology at the new fleet took the factory offline until the change was reverted. The fix (`2d7e45332`) left one definition, imported by both users.

9.6. When the Tests Fail

A red verdict means the candidate was evaluated and rejected. The gate execution is `red`, the attempt ends at `gate_red`, and nothing is pushed. An attempt is never reopened: a retry is a new attempt against whatever `main` has become. The red carries evidence, a bounded failed-shard summary in the verdict (`failed_shard_diagnostics`, added in `6f686d685` after early reds arrived with empty per-shard detail) and persisted failing node ids (`c10db11d0`), readable with `yesod refinery gate-report`.

One kind of red looks like a bug and is the gate working: semantic staleness. A branch written against an older `main` can merge cleanly and still fail, because it carries a test that pins behavior `main` has since changed. The candidate is the merge, so the gate sees the interaction the branch never tested. The remedy is rework against current `main`.

Two incidents show that a red or green label is only as good as the thing that counted it.

Factory Floor — Red runs counted green (2026-09-19). pytest reports a failed subtest as a separate report that reuses its parent's node id, while the parent's own report stays `passed`. Every counter in the factory filtered on the ordinary report type. On one attempt a shard printed "2 failed, 1403 passed" and exited 1, while the durable totals said 11,264 passed and 0 failed (`fa31cdf77`). The first fix covered four counting surfaces. A second pass found three more readers of the same report log, and in the commit's words "each turned a red run green differently": a pool qualification recorded a failed node as passed, a shadow validator accepted the log as green evidence, and a flaky-test scorer scored it passed (`cfb66cf03`). Both fixes were committed on 2026-09-20 UTC. A counter that cannot see a class of result must not back a verdict, and the regression tests for both fixes include runs of the real pytest.

Factory Floor — The merge without a green gate (2026-09-11). Merge commit `8326d9a3c` says in its own body that it was "Merged by operator decision … WITHOUT a green gate attempt". Seven of eight shards had passed. The eighth failed one test unrelated to the change: a temporary-directory cleanup race, `Errno 39`. The body records that 2 of 152 job records on that demon carried the same error and that hash partitioning sends a given test to the same shard every time, "Roughly a 1.3% false-red rate for any union containing it". The note could not be re-gated, because the supported path back into the gate did not apply to a note in its state, and the fix for that defect was one of seven notes blocked behind this one. Stephen decided, and the reason was written into history. The lesson is about cost: with a long gate and no cheap way to re-run one shard, a one-percent flake rate becomes pressure to bypass the gate.

Two responses are on `main`. Known-failure labels in gate reports (`9d60427ca`) are advisory and do not change a verdict. A targeted retest of one binding (`628a9fe8b`) sits behind `targeted_retest_enabled`, which defaults to `False`; the repository does not show it enabled.

9.7. When the Gate Cannot Run

The second deviation is the one the contract exists for. A demon is unreachable, a role's job is lost, or the worker rejects the request. The tests did not produce a judgment.

`GateProviderError` is documented in one line: "Provider transport or infrastructure failed; this is not gate red." In the union's reduction, any role record flagged `infrastructure_failed` or in phase `lost` makes the whole execution `lost`, ahead of every other outcome, even if other roles reported failures. The controller maps `lost` to the attempt state `infrastructure_failed`, and `transition_attempt_state` in `refinery_control.py` stamps a gate-finished time only for `gate_green` and `gate_red`, with the comment that the other terminal states "are terminal conditions rather than gate verdicts". A submission whose outcome is unknown raises `GateSubmissionOutcomeUnknown` and is resolved by lookup under the idempotency key.

Design Trade-off — Visible stall over silent fallback. Running the suite on the controller when the fleet is down would keep merges flowing, and it would change the interpreter, isolation and database assumptions behind the verdict without anyone seeing it. Yesod stalls the lane instead. On the Sefer provider this is structural: only the pinned command is accepted, and the rollout runbook says, "Rollback never runs a serial full suite."

The cost is that the lane's availability depends on the fleet's. The autumn tested that repeatedly.

A deadline shorter than the gate. The controller reconciles each gate against a per-lane deadline, `gate_reconciliation_seconds`. It was 3,600 seconds while the gate had grown to 44 to 54 minutes, so healthy attempts expired before their gates could report. The value in the reviewed document was raised to 5,400 (`cb4e90efc`, approved 2026-10-06) and to 7,200 on 2026-10-12 (`3bc462864`), and the worker's own per-role limit went from 90 to 120 minutes on 2026-10-11 (`ROLE_DEADLINE_SECONDS`, `46e9a7db2`). A timeout sized for an earlier workload fails healthy work.

A submission that never launched. According to the agents' working notes for 2026-09-09, gate hosts were stopped while a submission was in flight. The gate execution had no provider job id, its deadline passed without being reaped, and nothing merged for four hours. Chapter 8 tells this incident and the stall sentinel that followed it.

Reds that were the host. On the evening of 2026-09-17, by the agents' working notes, three consecutive rounds went red only on timeouts in tests unrelated to the change, on different shards each time, ninety minutes after a green on the same fleet. The storage pool behind the test databases was about 90 percent full. The diagnosis recorded in commit `9a4c22708` found a second cause: PostgreSQL writes 8 KiB pages, the database containers had inherited a 128 KiB record size from the pool, and every page write became a 128 KiB read-modify-write, "5-17 MB/s of random I/O per gate database". Test database birth now sets a 16 KiB record size, reads it back, and refuses any other outcome. These rounds were recorded as red verdicts and were statements about the host. The agents' rule since then: a red made only of timeouts in unrelated tests is re-requested at most once, and then the host is examined.

The host itself. On 2026-09-27 at 17:52Z `sefer` hard-reset under memory exhaustion, according to the agents' working notes. Gates stayed stopped after the reboot and every VM came back with new SSH host keys (Chapter 12). With one hypervisor, one host fault stops every Yesod merge.

9.8. Elasticity: What Is Live and What Is Staged

A gate that takes most of an hour, on a refinery lane that is strictly serial, bounds how many merges a day can hold. October produced a large amount of code to relieve that. Most of it is on `main` and not in service.

> **Proven Where — as of 2026-11-19.** *Live for the Yesod lane, per the reviewed controller document and the rollout runbook on `main` (last edited 2026-10-12 and 2026-10-22):* the eight-by-four union on profile `sefer-parallel-py3146-8x4-v1`, hash partitioning, one on-premises pool, a 7,200-second reconciliation deadline. *On `main`, staged, with runbooks that say the operator steps have not been performed:* duration-weighted partitioning (profile `...-duration-v2`, algorithm `median-lpt-duration-v1`) and its multi-root successor v3, which also collects `ops/**/tests`; the dynamic chunk scheduler and elastic 8 to 16 shards (`SEFER_MIN_SHARD_COUNT`, `SEFER_MAX_SHARD_COUNT`, `refinery_gate_chunks.py`), whose runbook is headed "operator-held"; the right-sized safety island host (2 cores instead of a shard's 8), "documented but not executed"; the headroom advisory that would recommend growing or shrinking the fleet, "advisory and disabled"; one-binding targeted retests, off by default. *On `main`, never run against a real cloud account as far as the repository records:* the Fargate transport for the container provider ("offline-accepted") and the Fargate Spot burst pool with its budget ledger ("offline review tooling only"). *Not enforcing:* the pre-gate, a cheap check before the union (Chapter 8), whose activation runbook carries the status line "procedure only; qualification and activation have not been performed" and which has no mode set on any lane in the reviewed document. *Not verified:* whether the live fleet today was born from gate fleet generation 5 or 6, and whether any staged item was activated after 2026-10-22, the last edit to the relevant documents.

Two related mechanisms are also on `main`. A lane may have up to three gate pools (`MAX_GATE_POOLS` in `refinery_pool_config.py`), each with its own nine-role topology, and an experimental second pool was built on `seykhl` in September; the reviewed document records one pool for the Yesod lane. And `yesod refinery gate-env` (`refinery_gate_env.py`, `451a8bcd7`; the verb in `463774a4a`) derives an identity for the gate environment from the files of one exact Git commit, without touching the database or the live fleet. That addresses part of the August edition's open question about which gate code produced a given verdict, and the module states its own limit: "KNOWN GAP v1: template provenance is not in gate_env v1".

Through the autumn the cost of the contract was throughput. No record on `main` shows a broken candidate landing through a green gate, though that is a reading of the chronicle and not an audit, and for a time the counters behind the verdict could not see failed subtests. The one commit known to have reached `main` past a red gate did so by a recorded human decision, and when `main` was left red on 2026-09-14 the cause was two pull requests merged outside the gate (Chapter 1). A green union lets the controller push the exact candidate, and the note's code is on `main`. Whether any machine is running that code is a separate fact, and Chapter 10 (Merged Is Not Deployed) starts there.

Source Notes

Claim or section	Tag	Anchor
States, providers, protocol	[code]	`refinery_state.py` (`GateExecutionState`); `refinery_gate_provider.py` (`GateProviderName`, `GateProvider`, `GateProviderError`)
Broker and idempotency	[code]	`refinery_gate_broker.py` (`GateBroker`, `GateSubmissionOutcomeUnknown`); `gate_execution_idempotency_key`; `sefer_job_id`
Who uses which provider	[deploy] 2026-10-12, 10-22	`controller-reviewed-document.json`; `docs/refinery-fleet-gates.md`; `ops/yesod-dispatch/README.md`; `19f764260`, `5a35c087d`, `f461ae0f9`
Why the gate came home	[deploy] 2026-08-28	note `ys-yes-wmuw` in `docs/tools/yesod.md`; `f56a845ac`, `1964730c3`; cost motive [inferred]
Pinned command, no fallback	[code]	`CANONICAL_PYTEST_COMMAND`; `LaneControllerConfig`; rollout runbook sections 1, 2, 8
Timeline to DG5-only	[deploy]	`64ebca833`, `2e77ba6b9`, `46ec1d209`, `9f1ad55b8`, `2584d2844`; `g1-cold-recovery-2026-09-16.md`
Union trace	[code]	`refinery_gate_capacity.py`, `refinery_gate_partition.py`, `refinery_sefer_gate.py`; the stager in `ops/sefer-gate/bin/`
Durations	[deploy] 2026-09-07, 10-06	`ops/yesod-birth/gate-profiles.json`; `cb4e90efc`; cause of growth [inferred]
Forced-command boundary	[code]	`ops/sefer-gate/sshd_config.d/`; `yesod-sefer-gate-forced-command` docstring; `sefer-forced-command-rollout.md`
Credentials on tmpfs	[code], [notes] 2026-09-09	`sefer_gate_credentials.py`; `SeferCredentialSynchronizer`; `sefer-gate-credential-lifecycle.md`; `a69954af0`
Preflight; hostname outage	[code]	`preflight-sefer-gate-fleet.py`; `SeferPoolReadinessProbe`; `ffaadbfe0`; `2d7e45332` body
Red evidence	[code]	`failed_shard_diagnostics`; `6f686d685`, `c10db11d0`; `refinery_cli.py gate-report`
Red runs counted green	[code]	`fa31cdf77`, `cfb66cf03` bodies
Merge without a green gate	[deploy] 2026-09-11	`8326d9a3c` body

Claim or section	Tag	Anchor
Lost is not red	[code]	`SeferGateProvider._observation`; `target` map in `refinery_controller.py`; `transition_attempt_state`
Deadlines	[code], [deploy]	`cb4e90efc`, `3bc462864`, `46e9a7db2`; `ROLE_DEADLINE_SECONDS`
Never-launched submission	[notes] 2026-09-09	agents' working notes; Chapter 8
Record-size reds	[code], [notes] 2026-09-17	`9a4c22708` body; agents' working notes
Host reset	[notes] 2026-09-27	agents' working notes
Elasticity status	[deploy] as of 2026-10-22	rollout runbook status line; the per-feature runbooks in `docs/runbooks/`
Gate environment identity	[code]	`refinery_gate_env.py`; `451a8bcd7`, `463774a4a`
Pools	[code]	`MAX_GATE_POOLS`, `GatePoolConfig` in `refinery_pool_config.py`

Not verified:

- The exact date the Sefer provider became authoritative for the Yesod lane (between 2026-09-02 and 09-04 by commits; no receipt on `main`).
- The full stated reason for leaving MicroVMs; the note body is truncated in `docs/tools/yesod.md`.
- Whether the live gate fleet on 2026-11-19 is gate fleet generation 5 or 6, and whether any staged elasticity item, the forced-command re-pin window, or the pre-gate was activated after the runbooks were last edited.
- Whether `ROLE_DEADLINE_SECONDS` at 120 minutes is installed on the live demons; the constant is in the worker on `main`, and installing a worker is a separate window.
- The cause of the gate's growth from about 11 minutes per shard to 44 to 54 minutes.
- The 2026-09-09 never-launched submission, the 2026-09-17 storage figures and the 2026-09-27 reset time rest on the agents' working notes only.
- That `8326d9a3c` is the only commit to reach `main` past a red gate is a reading of the incident chronicle, not an audit of merge history.
- The fleet preflight's individual checks were taken from the research brief and a keyword check of the script on `main`; the script was not re-read line by line.
- The date of the attempt in "Red runs counted green" is not in the commits; 2026-09-19 is the authors' local date of the first fix, which is 2026-09-20 in UTC.
- That no broken candidate landed through a green gate is inferred from the incident

chronicle.

Part III.

From Merge to Running Code

10 Merged Is Not Deployed

A green merge proves that source reached `origin/main`. It proves nothing about which code any process is executing. This chapter covers the distance between those two facts in a factory that deploys itself: where the distance is measured, the three ways Yesod tried to close it by changing its hosts in place between August and October 2026, why each of them kept leaving merged code unrun, and the states the factory grew to say so honestly.

The reader should finish with four things: the list of installs that drift apart, the reason "restart on the new code" is harder than it sounds when the deployer is inside what it deploys, the exact meaning of the attempt states `deploying` and `merged_not_deployed`, and the reasoning that led Stephen, on 2026-10-13, to stop mutating long-lived hosts. Chapter 11 (Deploy Is a Host Swap) picks up from that decision.

10.1. The Gap

For the example command followed since Chapter 1 (Follow the Change), a successful push means its source is on `origin/main`. The `yesod` program on every machine is still the one that was installed there last. A long-lived service is still the process it was started as. The note can be finished in the source lifecycle while the feature is absent from every running program.

For most projects closing that gap is small and local: copy an executable, restart a service, undo it if it goes wrong. Yesod deploying itself is a different problem because failure is partial. A migration applies and one host never gets the matching code. A service restarts and comes back reading its old tree, and it answers requests, so nobody notices. Half the hosts move, a health check fails on the next one, and two releases are live at once. And the thing doing the deploying is one of the things being deployed.

The most expensive property of the gap is that it is silent. Runner seats execute an immutable release tree under `/srv/yesod/releases/<sha>`, selected by the `/srv/yesod/current` link, and never `main` itself. Planning runs inside the runner process from that same tree. The agents' working notes for 2026-09-10 and 2026-09-11 record the consequence. A fix to the runner watchdog merged with a green gate and was running in no process. A fix to a plan validator that was blocking six notes was inert in the same way until deployed: a re-plan after the merge fails with exactly the message it failed with before. The symptom after the merge is identical to the symptom before it, which reads as "the fix did not work" and invites reopening a note that is correct. The same notes record Stephen's remark at the time: "once these things land they still need to be deployed."

10.2. Four Installs That Drift

A Yesod deployment is several installs, and they move at different times. The module `release_skew.py` names four of them in its docstring:

Surface	Where	What runs from it
Operator CLI	pompom	Every command an operator types
`current`	The controller	Refinery, scheduler, exporter
`ops-current`	The controller	The CLI the controller uses to read gate results
`current`	Each build seat	The runner and every agent it starts

The same docstring records the measurement that motivated the module. On 2026-09-20 the seats were 0 commits behind `main`, the controller 35, and the operator CLI 529, "and nothing reported any of it". A stale surface does not fail. It answers with old behavior.

The read-only command `yesod release-skew` (commit `7bb582e12`, 2026-09-21) reports each surface's release and its distance from `origin/main`. Two rules are built into its types. The release is the install path: `release_sha_from_path` reads the 40-hex directory name, because an exported tree has no `.git` and the version literal never changes. And an unknown is never a zero: `SurfaceObservation` and `Distance` cannot be constructed unidentified without a reason, so a host that did not answer is reported with a status such as `unreachable`, `unknown_release` or `unknown_distance`, and only a measured distance of zero is reported `current`. The companion commit `308628d7d` made the scripts that move a release end by quoting the measured distance, and states the rule in its message: "`deployed` is a claim, `0 behind` is a measurement."

> **Operator Check — How far behind is each install?** Run `yesod release-skew` from the operator machine. It executes `readlink` on the remote hosts and `git rev-list` locally and writes nothing. `--json` emits the same document, `--strict` exits 1 when any row is not `current`, and `--local-only` skips the network. A row marked `unreachable` is a finding in its own right.

Gate VMs carry a fifth copy, a forced-command script placed at birth, which Chapter 12 (Born, Not Repaired) covers.

10.3. August: One Transaction Across Four Hosts

The first answer, described at length in the August edition of this book, was the **promotion**: one durable, restart-safe transaction that moved every host from a `prior_sha` release to a `candidate_sha` release, or refused and left the fleet where it was. At the time the fleet was the control host (then called the Conductor, on host `yesod-dispatch`) and three coding runners.

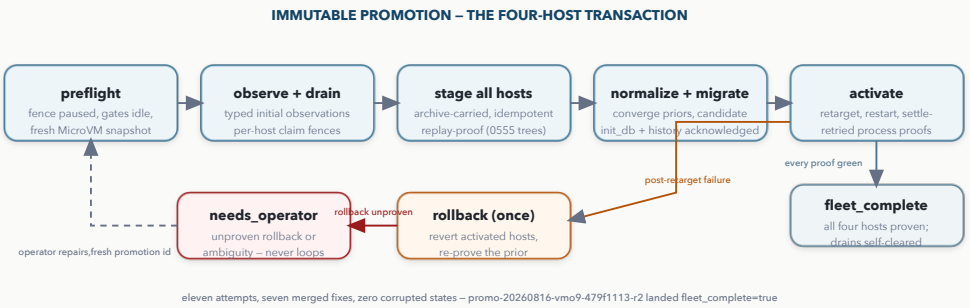

Figure 10.1.: The August four-host promotion transaction, retired as the routine path and shown here as history.

Its vocabulary survives in today's code and is worth keeping. An **immutable release** is the content of one commit staged read-only at `/srv/yesod/releases/<sha>`, with a `RELEASE_SHA` marker and a `RELEASE_COMPLETE` marker written last, so a half-copied tree is never mistaken for a release. The `current` link selects the release that services run, and the separate `ops-current` link keeps the tooling that performs a deployment distinct from the code being deployed. Every external act was journaled twice, an intent before the act and an observation after it, so a crashed worker could recompute its position without repeating a migration or a link flip.

The order was the design: observe every host, pause merge admissions (Chapter 8, The Merge Controller), drain the runners, stage the release on all four hosts before activating any, migrate once, then flip `current` and restart host by host with proof that each restarted process was executing the candidate. A failed proof after activation caused one bounded rollback to the prior release. A failure before any pointer moved ended `needs_operator`. The first real executions were an eleven-attempt campaign in mid-August, told in Chapter 15 (The Reliability Contracts), in which no attempt left a half-moved fleet.

That machinery is still on `main` under `yesod release-promotion` (`prepare`, `run-pending`, `hold`, `resume`, `status` and others), but it is no longer the routine path. The script `submit-release-promotion.sh` reads `YESOD_RELEASE_PROMOTION_MODE` with a default of `replacement`, and accepts `legacy` only as the alternative. The transaction was correct and safe. It was also something an operator started by hand, so merges accumulated between runs.

10.4. September: Seats Deploy Themselves, the Controller Does Not

In September the fleet changed shape. On 2026-09-16 a fresh VM, VM175, replaced the original controller VM by a hand-run handover; the runbook `controller-g1-handoff-2026-09-16.md` records a normal gate under the new controller passing at 20:27:01 UTC. The next day a whole-fleet release was done by hand inside one admissions pause: the controller in a 34-second stop, retarget and start window, then six runner seats (a canary first, then one seat

at a time), then the forced-command script on the nine gate VMs. Its record, `fleet-release-244a8ab1-2026-09-17.md`, ends with a plain statement of the gap: "No routine merge-to-fleet transport exists on VM175 yet; until it does, merges strand until an operator repeats this procedure."

Routine deployment was the answer to that sentence. A refinery lane whose configuration carries a `deployment` block does not finish at the push. With `deploy_after_push` set, the controller's `SETTLE_PUSH_APPLIED` action moves the attempt from `pushing` to a new active state, `deploying`, with reason `push_confirmed`, and a configured `deployment_stepper` is then polled on every tick. The stepper applies the candidate to the runner seats and asks an independent observer whether it arrived. The observation covers three surfaces, named by `SURFACE_NAMES` in `deployment_surfaces.py`: `runner_releases`, `sefer_birth` (the files on the hypervisor from which new VMs are born) and `controller_policy` (the controller's live configuration compared with a reviewed copy). The plan of record for a combined window on 2026-09-18 adds the block to the live controller document (`combined-controller-window-2026-09-18.md`). It wedged that day (see below) and was turned off, and docs-only sentinels for re-activation followed on 2026-09-19 and 2026-09-20 (`routine-activation-sentinel-2026-09-19.md`, `routine-activation-window-2026-09-20/SENTINEL.md`). The second of those says that, when it was written, the routine apply had never completed in production.

Once the block was active, runner seats took each merged release on their own. No file on `main` records the first completed routine apply; the evidence is the `release_skew.py` measurement of 2026-09-20 (seats 0 commits behind) and the working notes from October. The controller did not. Its two links still moved only when an operator ran a promotion window by hand. The factory had automated deployment for the hosts that run agents and left the host that decides what merges on whatever release it had last been given.

10.5. Why Mutating a Long-Lived Host Kept Failing

The in-place path failed in the same few ways for six weeks. The incidents below are grouped by cause.

10.5.1. The deployment step wedged itself

> **Factory Floor — 4,000 bytes (2026-09-18).** The command stepper truncated every command stream to 4,000 bytes before parsing it as JSON. A routine observation was about 40 KB, so it was invalid on every tick, the apply never ran, and the attempt held the Yesod refinery lane in `deploying` with a self-renewing lease until an operator unwedged it by hand. Commit `b014de872` made the stepper parse complete streams and gave `deploying` a budget, `deployment_settle_seconds`, measured from `deploying_at`. A check that sits between a merge and the next merge needs a deadline, because without one a broken check stops all merging.

A day later a test run broke the release tree itself. On 2026-09-19 at 23:36Z a pytest run under the deployed release's interpreter wrote sixteen `__pycache__` directories and five `.pyc` files into the sealed tree. The immutability check worked: every stage verb on the controller

refused from that moment, which blocked promotion until an authorized repair. Commit `2237c49e5` added `controller_pytest_guard.py`, which refuses any pytest invocation whose targets, working directory, `--basetemp`, `HOME` or `TMPDIR` resolve under the release root or either link, and changed the stage verb to report every offending path in one refusal (the burst was 118).

On 2026-09-21 an authorized window added two refinery lanes to the controller's live configuration document. The `controller_policy` probe compares the live document with the reviewed document inside the controller's installed release, so 22 values read as drift and every Yesod-lane attempt parked in `deploying`. Commit `7e8b9def6` brought the reviewed document up to the live shape, and its message records the trap: the merge "alone changes nothing on VM175 until a release carrying it is promoted and the controller restarted". The fix for a deployment wedge was itself merged and not deployed.

10.5.2. Code arrived before its schema

> **Factory Floor — Runners ahead of the schema (2026-10-07 and 2026-10-13).** Routine deployment applies no migrations; they landed only at a controller promotion. Twice in one week a merge rolled the seats onto code that needed a migration the database did not have. The agents' working notes record about 19 hours with the fleet down on October 7 and 8. On October 13 the new code inserted columns that did not exist, `_start_run` swallowed the exception, and every dispatch failed with "graph evidence persistence failed at ordinal 0" until guarded statements were applied by hand. The rule the agents adopted was expand and contract: code on a runner path must tolerate the schema from before its own migration. These two accounts rest on the working notes; no commit on `main` carries either incident in its message.

10.5.3. Replaying old schema statements

Three incidents came from the way `init_db` healed a schema whose stored checksum differed from the code's. During one production upgrade a heal replayed 201 baseline statements, and a no-op `ADD COLUMN IF NOT EXISTS` still requested an exclusive lock while a backup was reading the table (runbook `baseline-ddl-replay.md`, which does not date the incident). Commit `524d10f61` (2026-09-21) routed the replay through `GuardedDDL`, which skips a statement when a read-only catalog probe proves its object exists (9 of 201 statements then ran on a materialized catalog, per the commit).

The heal also replayed every forward migration. On 2026-09-25, and again on 2026-10-14, a replayed migration re-added a CHECK constraint with an older, narrower value set and failed on live rows that a later migration had made legal (working notes for both dates). Both transactions rolled back. The second aborted a promotion window ten minutes after a full test pass of the same payload had come back clean, and that pass could not have caught it, because every test starts from an empty schema. A static scan afterwards found six such chains.

10.5.4. Something else migrated production

> **Factory Floor — The nightly job that migrated production (2026-10-15).** At 08:23Z a scheduled CI workflow that builds daily summaries ran `main`'s code against the production database with no flag telling it to leave the schema alone, and applied a migration outside any window. The working notes put the consequence in one sentence: the rule "migrations only at promotion" was false in practice. Any migration merged to `main` reached production within about a day. The job was fenced the same day (ys-yes-kz8o, commits `34e1d4085` and `968bbb34a`), and the class was closed two weeks later by making migration opt-in everywhere.

10.6. A State for the Gap

While these were being fixed one at a time, the refinery acquired a way to say what had happened. Commit `b014de872` had sent an overdue `deploying` attempt to `needs_operator`. Migration `0092_refinery_merged_not_deployed_ys_yes_j0hc.py` (2026-10-04) added a more exact terminal state to `refinery_attempts_state_check`: `merged_not_deployed`.

The only edge into it is from `deploying`. In `_run_deployment`, a stepper answering `PENDING` or `UNKNOWN` past `deployment_settle_seconds` ends the attempt `merged_not_deployed` with reason `deployment_settle_timeout`, and a `FAILED` answer ends it with reason `deployment_failed` or one of the named `RUNNER_ROLLOUT_INVENTORY_CONDITIONS`. The dataclass default for the budget is 86,400 seconds; the reviewed controller document on `main` sets `settle_seconds` to 1800. The comment beside the setting gives the purpose: such an attempt "frees the lane instead of holding the merge turn forever".

A recent real example: the note ys-yes-gn4b (fixes for the first stage of a successor controller's boot) was admitted in a merge window that the working notes place on 2026-11-18. Its attempt went `requested` → `claimed` → `preparing` → `gating` → `gate_green` → `pushing` → `deploying` → `merged_not_deployed`, and `origin/main` became the candidate commit `16606a42`, the snapshot this edition is written against. That commit is stamped 2026-11-17 19:16Z, a time derived from its source commit, not the time of landing. The sequence is from the agents' working notes, which omit `deploying`; the transition map in `refinery_state.py` allows no other way in. The code those commits fix is on `main` and is running on no controller.

Three things follow from the terminal row.

First, the controller's `_merged_not_deployed` opens an operator item whose recommended actions are `request_deployment_retry` and `inspect_deployment_observer`. (When the reason is one of the runner rollout inventory conditions, the first recommended action is `reconcile_runner_rollout_inventory` instead.) The row itself can never be edited: the trigger function `refinery_reject_terminal_attempt_mutation` includes the new state. A retry is a new attempt that goes from `claimed` straight to `deploying` (reason `deployment_retry_started`) without preparing or gating again, and `request_retry` in `refinery_control.py` refuses to create it without the candidate recorded on the old one: "merged_not_deployed retry requires the preserved candidate SHA".

Second, the note is told. `_project_merged_not_deployed` in `refinery_intake.py` writes the note's status as `blocked`, sets the `blocked` disposition with a JSON reason naming the attempt, the candidate and the observer evidence, and sets the `merge_queue` row to `status='held'` with `gate_state='merged_not_deployed'` and a `refinery_v2_projection` flag carrying `retry_required: true`.

Third, a later deployment can finish the story without rewriting it. `refinery_settlement.py` is written to prove that the recorded candidate reached an applied deployment and that the source branch is identical to the target, and then to repair the note, queue and inbox projections while leaving the terminal attempt as it is.

The projection has a cost in legibility. A note at `blocked` normally means the work could not go forward, and a note projected from `merged_not_deployed` is `blocked` with its code on `origin/main`. The disposition reason, or the attempt, has to be read before such a note is treated as failed. For the question "is it merged", both terminal states count: `note_settlement_read._MERGE_TERMINAL_STATES` is `("succeeded", "merged_not_deployed")`.

10.7. October: The Controller Tries to Promote Itself

The agents' working notes for 2026-10-05 record the controller sitting on an old release while `main` carried merged fixes to it, and Stephen's comment: "it is not good when the factory itself goes behind." The response was controller self-promotion (ys-yes-7dzm, commits `032800495` and `b6c7d2423`, 2026-10-06; migration `0093_controller_self_promotion_ys_yes_7dzm.py`; runbook `controller-self-promotion.md`).

The design kept the in-place move and removed the operator. After the seats and the birth surface reach the candidate, the stepper enqueues one controller promotion and stays pending. A standing service waits for an idle gap (no active gate, no controller planning lease, no existing admissions pause), then walks the steps of `ControllerSelfPromotionStep` in order: `admissions_paused`, `controller_units_stopped`, `database_migrated`, `config_reconciled`, `release_links_updated`, `controller_started`, `controller_tick_observed`, `policy_proved`, `support_services_started`, `admissions_resumed`. A failed forward step restores the prior release and ends `rolled_back`.

It did not close the gap. According to the working notes for 2026-10-10, from the promotion of the release that carried this work on 2026-10-09, every routine deployment sat in `deploying` for about 75 minutes and then ended `merged_not_deployed` with reason `deployment_failed`. The new per-service check resolved each controller service's release by reading the process's working directory under `/proc`, and on that hand-built controller one service ran as a different user, so the read was denied. Seats still received each release. The fix (commit `619f866f4`, ys-yes-3zfv, 2026-10-10) resolves the release from the process command line, and like every controller fix it needed a hand-run promotion before it could take effect. The notes read for this chapter do not record a self-promotion completing unattended in production.

The numbers that followed are from the working notes and were not measured from source. In the 24 hours around 2026-10-13, 15 of 15 merges ended `merged_not_deployed` on settle timeouts. By 2026-10-14, 51 merges were parked in that state, and they exposed a controller guard that assumed `succeeded` meant "the latest pushed target" and so deadlocked the refin-

ery lane (told in Chapter 8). The controller that began serving on 2026-11-03 carried all 139 such changes, according to the notes for that date.

10.8. The Dependant Problem

A note's `depends_on` field holds the ids of notes that must land first. The runner's claim predicate, `_note_dependencies_satisfied_sql` in `claimable.py`, treats a prerequisite as satisfied only when its status is `done` or `superseded`. A merged-but-undeployed prerequisite is `blocked`. So a follow-up to work that is already on `main` cannot dispatch, and when every merge ends `merged_not_deployed`, that is every follow-up.

Nothing on `main` resolves this in the dispatch path. `note_settlement_read.py` computes what a merge-aware rule would conclude (`MERGED`, `NOT_MERGED`, `NOT_ON_MAIN`, `REVERTED`, `UNKNOWN`), and its docstring says it "deliberately does not participate in any dispatch decision".

The workaround is a convention, recorded in the working notes on 2026-10-24 and held by the coordinating agents, with no code behind it. The dependency edge is cleared before the follow-up is planned, once the prerequisite is proven an ancestor of `origin/main`. In its place, the first acceptance line of the follow-up's first commit-bearing child is a search for a literal string the prerequisite introduced, run before any edit. If the string is absent the child changes nothing and finishes with the structured outcome `blocked_on_dependency` (Chapter 7, Governed Agent Execution). Each cleared edge is announced on the bulletin board. The check that the database would have made is moved into the task and performed by the worker against its own base commit.

10.9. Schema Changes Become Opt-In and Missing-Only

The schema incidents produced three changes that hold on `main`.

Applied migrations are never executed again. Since ys-yes-kavo (`9937e15f1`, `9e99dd2e4`, 2026-10-14), `_apply_forward_migrations` reads the `schema_migration_applied` table and runs only ids that are missing, even when a checksum mismatch sets `force_rerun`. Its docstring gives the reason: "an applied migration can never narrow a live constraint by replaying old SQL."

Migration is opt-in. Since `c3451be82` (2026-10-30), `init_db` runs no DDL by default. It calls `_verify_schema_version` and raises `DatabaseConfigurationError` when the catalog is behind the code. Only a process started with `YESOD_ALLOW_MIGRATE=1` may migrate, and `YESOD_SKIP_INIT_DB`, which the runner sets for dispatched agents, wins when both are present. Before this change the error text for a behind catalog advised running from a checkout "without YESOD_SKIP_INIT_DB" (`db.py` at the parent of `c3451be82`), which would have migrated production from whatever checkout the reader happened to be using (the hazard is recorded in the working notes for 2026-10-16).

Automatic deployment refuses migrations. In the replacement design, `ReplacementMigrationCoordinator.complete_automatic` raises `AutomaticMigrationRefused` for any candidate

with a pending migration (commit `8001389bc`, 2026-10-27). Schema changes go through a hand-run window. This is a retreat from the August transaction, which ran the migration itself, and it was chosen after the incidents above, in which a migration reached production, or failed to, on a path nobody had reviewed.

These fixes created a skew of their own. `CURRENT_SCHEMA_VERSION` is computed from the length of the migration registry, so from the moment a migration merges, `main`'s code wants a newer catalog than production has, and a current build of the CLI refuses to run against production until the window that applies it. The working notes for 2026-10-16 record operators and agents using pinned older builds for that reason.

10.10. The Decision of 2026-10-13

By mid-October the pattern was plain to the people and agents running the factory. Each failure had a specific cause and a specific fix, and each fix was correct. What the failures shared was their setting: a host that had been running for weeks, with accumulated files, service users, live configuration and half-released drains, being changed while it worked.

The agents' working notes for 2026-10-13 record Stephen's rule: "ALL deployment should be basically stopping one VM/LXC and starting a different one, with a bit of handover." The same note lists what the rule removes (among them settle timeouts, read-only filesystem errors, bytecode in sealed trees, a trap between drains and pauses, and no clean rollback) and records the hand-run promotion as a stopgap from that day. Three weeks earlier, on 2026-09-22, he had said the same of runners: "cattle, not pets".

In outline, a new controller VM is born from a template at the candidate release, proves itself while holding no authority, and is handed authority by a compare-and-swap in the database, after which the old VM stops and is kept for one release. The code for this began landing the same day (ys-yes-mabj) and the routine deployment step on `main` now submits a replacement intent and leaves the serving host untouched (`apply_candidate` in `routine_deployment.py` requires `serving_host_mutated` to be false). The attempt then stays in `deploying` until the intent is terminal. Chapter 11 describes that design and its status, including a reading of the code, never exercised, in which that wait keeps a merge-triggered handover from proving quiet.

> **Proven Where — as of 2026-11-19.** The August four-host promotion ran in production and is retired as the routine path. Runner seats taking each merged release was activated in production in windows on 2026-09-18 to 2026-09-20 (runbooks on `main`, which are plans and sentinels and do not record the first completed apply); the working notes for October record seats receiving releases. The `merged_not_deployed` state and its note projection are on `main` and are the ordinary outcome of a production merge according to the working notes. Controller self-promotion is on `main`; no unattended production run is recorded in the sources read. Every controller change so far was run by hand. The merge-triggered automatic swap that would move a `merged_not_deployed` note to `done` without an operator has not happened in production.

Source Notes

Claim or section	Tag	Anchor
The Gap: pinned release, planning inside the runner, identical symptom	[notes]	agents' working notes, 2026-09-10/11
Four installs; 0 / 35 / 529; status names	[code]	`release_skew.py` docstring, `SURFACE_KINDS`, `STATUSES`; `7bb582e12`, `308628d7d`
`yesod release-skew` flags	[code]	`cli.py release_skew`; `7bb582e12`
August promotion vocabulary and order	[code]	`release_promotion_cli.py`; migration `0063_release_promotion_state.py`; August edition ch. 10
Retired as routine path	[code]	`submit-release-promotion.sh`, `YESOD_RELEASE_PROMOTION_MODE`
VM175 handover; 34 s window; "No routine merge-to-fleet transport"	[deploy]	`controller-g1-handoff-2026-09-16.md`; `fleet-release-244a8ab1-2026-09-17.md`
`deploying`, `deploy_after_push`, three surfaces	[code]	`refinery_controller.py`; `deployment_surfaces.py` `SURFACE_NAMES`
Block installed, sentinels	[deploy]	`combined-controller-window-2026-09-18.md`; `routine-activation-sentinel-2026-09-19.md`; `routine-activation-window-2026-09-20/SENTINEL.md`; `b014de872`
4,000-byte truncation	[code]	`b014de872`
Sealed-tree pollution, 118 paths	[code]	`2237c49e5`; `controller_pytest_guard.py`
Config edit wedge, 22 values	[code]	`7e8b9def6`; `deployment_controller_probe.py`
Runners ahead of schema; 19 hours	[notes]	agents' working notes, 2026-10-08, 2026-10-13
Heal versus backup, 201 and 9 statements	[code]	`524d10f61`; `baseline-ddl-replay.md`; `GuardedDDL` in `db_baseline_ddl.py`
Replay narrowing, six chains	[notes]	agents' working notes, 2026-09-25, 2026-10-14; fix `9937e15f1`
Nightly job migrated production	[notes]	agents' working notes, 2026-10-15; `34e1d4085`, `968bbb34a`
`merged_not_deployed` state, reasons, operator item	[code]	migration `0092_...j0hc`; `_run_deployment`, `_merged_not_deployed`
Settle budget 86,400 / 1800	[code]	`deployment_settle_seconds`; `controller-reviewed-document.json`

Claim or section	Tag	Anchor
ys-yes-gn4b attempt	[notes]	agents' working notes, 2026-11-18; commit `16606a42` (stamped 2026-11-17 19:16Z, from its source commit); transition map in `refinery_state.py`
Retry contract; immutability	[code]	`refinery_control.py request_retry`; `BEGIN_DEPLOYMENT_RETRY`; trigger function `refinery_reject_terminal_attempt_mutation`
Note projection	[code]	`refinery_intake.py _project_merged_not_deployed`
Later settlement	[code]	`refinery_settlement.py` module docstring
Controller self-promotion design	[code]	`controller_self_promotion.py`; `controller-self-promotion.md`; `032800495`, `b6c7d2423`
Probe permission failure, about 75 min	[notes]	agents' working notes, 2026-10-10; fix `619f866f4`
15 of 15; 51; 139	[notes]	agents' working notes, 2026-10-13, 2026-10-14, 2026-11-03
Dependency predicate; shadow settlement read	[code]	`claimable.py _note_dependencies_satisfied_sql`; `note_settlement_read.py`
Dependant workaround	[notes]	agents' working notes, 2026-10-24
Missing-only replay; opt-in migration	[code]	`db.py _apply_forward_migrations`, `init_db`, `_migrate_allowed`, `_verify_schema_version`; `c3451be82`
Automatic refusal of migrations	[code]	`replacement_migration.py AutomaticMigrationRefused`; `8001389bc`
Schema version skew after a migration merges	[notes]	`CURRENT_SCHEMA_VERSION` in `db.py`; agents' working notes, 2026-10-16
Stephen's rules of 2026-09-22, 2026-10-05, 2026-10-13	[notes]	agents' working notes, those dates
Routine step submits a replacement intent	[code]	`routine_deployment.py apply_candidate`, `_submit_replacement`

Not verified:

- Whether controller self-promotion ever completed unattended in production. The working notes read for this chapter record it blocked and worked around; they do not record a completed run.
- The reason code on the ys-yes-gn4b attempt, and whether it passed through `deploying` as the code requires (the notes list `pushing` then `merged_not_deployed`).
- The counts 15 of 15, 51 and 139, the 75-minute figure, the 19-hour outage and the six constraint chains are from working notes only.

- The date of the first routine apply that completed in production. The runbooks on `main` for 2026-09-18 to 2026-09-20 are plans and sentinels.
- The date of the heal-versus-backup incident; `baseline-ddl-replay.md` does not give one, and 2026-09-21 is the date of the fix.
- `refinery_settlement.py` was read at its module docstring only; whether it has run in production was not checked.
- Which production windows applied which migrations, and whether the live controller document still carries the `deployment` block with `settle_seconds` 1800, were not checked (no host access).
- The CLI verb an operator uses to request a deployment retry was not identified; the chapter describes the contract in `refinery_control.py` only.
- The 2026-10-07/08 incident has no commit on `main` under its own note id; the account follows research brief 10 item 26.

11 Deploy Is a Host Swap

This chapter describes how Yesod is designed to deploy itself after a merge: by building a new controller machine, proving it, handing it authority through one database compare-and-swap, and switching the old machine off. The reader will learn the durable state behind a swap, the fifteen steps in order, the single step after which automation refuses to go back, and the rules the code enforces at each point.

The reader will also learn exactly how much of this has run. As of 2026-11-19 the design is on `main`, parts of it have run in production, and the handover at its center has never run in production. No replacement intent has reached `succeeded` there. Every change of controller so far was carried out by an operator. The last section of the chapter separates what is designed, what is merged, what was rehearsed and what is proven, and gives a dated account of the attempts.

11.1. Why a Swap

Chapter 10 (Merged Is Not Deployed) ends with the rule Stephen set on 2026-10-13: every deployment should amount to stopping one machine and starting a different one, with a short handover between them. Each failure of the earlier in-place path came from changing a long-lived host while it was serving.

The replacement design removes that mutation. The runbook `docs/runbooks/routine-deployment.md` states the rule for routine releases: deployment never retargets the release links, installs candidate controls, or restarts candidate services in place on the serving controller. The serving controller keeps running the release it was born with until the moment it stops. Everything new happens on a machine that has no authority yet, so a failure there costs a disposable VM and nothing else.

The price is a handover problem. Two machines exist at once, both able to run the merge controller, and exactly one of them may write.

11.2. The Vocabulary of a Swap

- **Predecessor** and **successor.** The serving controller and the machine that replaces it.
- **Replacement intent.** One durable row that says “this scope should move from release A to release B”. It has a phase and, at the end, a terminal disposition. Submitting an intent changes no host.

- **Target.** A host named by an intent, with its own state. A controller swap has two targets.
- **Effect.** One side effect on the world (a clone, a migration check, a leadership transfer, an address release), journaled before it is attempted and observed afterwards.
- **Authority.** A single database row naming the controller that may run the refinery, with a revision and a lease generation.
- **Stop guard.** The swap's own admissions pause plus the proof that nothing was in flight when it was taken.
- **Replacement driver.** The program that advances an intent by one pass. It is `yesod replacement drive`, class `ReplacementDriver` in `replacement_driver.py`.
- **Service addresses.** The fixed network addresses that runners and telemetry use to reach the controller. They move from predecessor to successor during the swap.
- **Binding.** The reviewed file on the controller that describes the environment a swap runs in (template, storage, time bounds). Its model is `ReplacementBinding` in `replacement_plan.py`.

> **Terminology Trap — four kinds of generation.** A *controller host generation* is a hand-numbered controller VM (g1 was VM175; g5 is the one serving in November). An *intent generation* is the counter in `replacement_intents.generation`, one per submitted candidate within a scope. A *lease generation* is `refinery_controller_authority.lease_generation`, the fencing number that the controller's refinery-lane writes are checked against. A *gate fleet generation* (dg5, dg6) labels the birth recipe a set of gate VMs was built from (Chapter 12, Born, Not Repaired). This chapter always says which one it means.

11.3. What Is Written Down

A swap crosses two machines and must survive either of them dying, so nothing about its progress lives in memory. Migration `0117_replacement_promotion_state_ys_yes_mabj.py` creates the tables.

Table	One row per	Holds
`replacement_intents`	swap	candidate, prior release, phase, deadline
`replacement_intent_heads`	scope	the current intent
`replacement_intent_attempts`	covered merge	intent to refinery attempt
`replacement_targets`	host in a swap	state, authority and stopped flags
`replacement_effect_journal`	side effect	kind, key, evidence state, digests
`replacement_stop_guards`	intent	the owned pause and its lease

An intent's `phase` follows one path, defined by `ReplacementPhase` and the map `_PHASE_TRANSITIONS` in `replacement_contract.py`:

```
planned -> birthing -> qualified -> ready -> pausing -> draining
        -> fencing -> activating -> (rolling_runners ->) settling
        -> succeeded
```

There are two other terminal phases. `needs_operator` is reachable from every non-terminal phase. `rolled_back` is reachable only from `fencing`, `activating`, `rolling_runners` and `settling`. The function `require_phase_transition` rejects a skipped or reversed edge in Python, and the trigger `trg_replacement_intents_fenced` rejects it again in the database.

A target moves through `ReplacementTargetState`: `planned`, `born`, `qualified`, `passive`, `serving`, `stopped_retained`, `destroyed`, with `failed` as the exit from any state before `destroyed`. The edge from `stopped_retained` back to `serving` is legal. That edge is what makes it possible to bring a retained predecessor back.

Each row of `replacement_effect_journal` has an `evidence_state`: `intended`, `outcome_unknown`, `confirmed_applied` or `confirmed_not_applied`. The driver claims an effect with `ReplacementStore.claim_effect_intent` before acting and records the result with `observe_effect`. The `idempotency_key` column is unique, so a pass that restarts after a crash finds its own earlier claim and does not act twice. A confirmed row cannot change afterwards (`trg_replacement_effect_journal_fenced`).

Authority lives in a separate table from an earlier migration (`0097_controller_authority_ys_yes_ectq.p`). `refinery_controller_authority` holds the owner's hostname and machine id, a `revision`, a `lease_generation` and a lease expiry. Every change appends a row to `refinery_controller_authority_events` with type `initialized` or `transferred`. A companion table, `refinery_controller_lane_leadership` (migration `0118_controller_leadership_fence_ys_yes_ma`) records a holder and a leadership generation for each refinery lane; a transfer requires every row to name the current owner at the current lease generation. Chapter 8 (The Merge Controller) describes how the controller checks this row on every write.

11.4. What Starts a Swap

After the refinery pushes a candidate to `main`, the attempt enters `deploying` and the controller calls its deployment stepper (Chapter 8). On `main` the default branch of `routine_deployment.apply_candidate` builds a submission plan from the binding file with `generate_submission_plan_from_binding` and calls `submit_replacement_intent`. It then checks two things about the answer: that it names the same candidate, and that its `serving_host_mutated` field is exactly `False`. Submission writes rows and touches no host. The command-line form, `yesod replacement submit`, exits with code 75 (`EXIT_NONTERMINAL`) to say that the intent exists and is not finished.

Merges arrive faster than swaps can run, so intents coalesce. `ReplacementStore.coalesce_intent` takes an advisory lock on the scope and reads the scope's head. If the head already names this candidate, the new attempt is added to its coverage and nothing else changes. Otherwise a new row is inserted with the next intent generation, the earlier intent (if it is not terminal) is marked with `superseded_by_intent_id`, and the earlier intent's attempt coverage is copied forward. Three merges that land while one swap is being prepared therefore end as one swap to the newest candidate.

The work itself is done by the driver. The unit `yesod-replacement-driver.timer` runs `yesod replacement drive` once a minute (`OnUnitActiveSec=1min`) as a oneshot service that systemd will not overlap. `ReplacementDriver.drive` is documented as one non-blocking pass without retry loops: it reads the current state, takes the next step if it can, and exits. If the binding file is absent the driver prints `replacement binding unavailable` and exits successfully.

Both units ship installed and disabled. The runbook `docs/runbooks/replacement-driver.md` says the operator enables the timer only after the reviewed binding is installed. So the chain from merge to swap is wired in code, and whether it is switched on is a fact about the deployment. According to the agents' working notes for 2026-11-11, the driver timer and service on the serving controller were being kept disabled at that date.

11.5. One Swap, Step by Step

The trace below follows one controller swap as it is coded on `main`. The agents' working notes describe the production attempts as hand-passed runs, with the intent submitted by command and not by a merge. The steps that produce a driver-born successor (3 to 5) have been entered there and never completed. Steps 6 to 15 have never run in production. Step numbers throughout this chapter are those of the table.

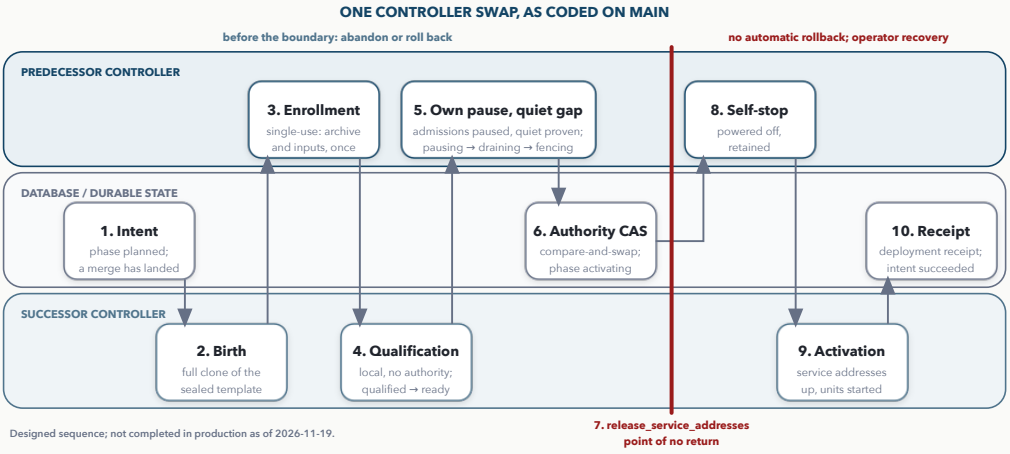

Figure 11.1.: A controller swap runs across the predecessor, the successor and the database, and the release of the service addresses is the point of no return. The figure condenses the fifteen steps of the table into ten.

#	Actor	Step	Intent phase
1	refinery	merge pushed; intent submitted	`planned`

#	Actor	Step	Intent phase
2	driver	pre-birth refusals checked	`planned`
3	driver	successor cloned from the template	`birthing`
4	driver, root helper	single-use enrollment	`birthing`
5	successor	install candidate, qualify itself	`qualified`, `ready`
6	driver	own pause, quiet gap proved	`pausing` → `fencing`
7	driver	provider handover to successor	`fencing`
8	driver	authority compare-and-swap	`activating`
9	driver	predecessor prepared to retire	`settling`
10	driver, root helper	service addresses released	point of no return
11	driver	handover receipt, self-stop	`settling`
12	successor driver	predecessor proved stale	`settling`
13	successor root unit	addresses up, services started	`settling`
14	successor driver	pause released, receipt published	`succeeded`
15	refinery	attempt settles	attempt `succeeded`

"Driver" without a qualifier means the driver running on the predecessor. Until step 12 the predecessor does all the driving, and the successor only answers.

11.5.1. Before any machine exists (steps 1 and 2)

Submission records the intent, its two targets, and two effects: `replacement_budget`, an immutable copy of the time bounds from the binding, and `replacement_predecessor_facts`. From then on the driver reads the recorded copy, so editing the binding during a swap changes nothing about that swap.

On its first pass the driver can refuse before it constructs a provider. Each refusal ends the intent in `needs_operator` with a typed `failed_step`: `replacement_protocol_version` if, for a swap that moves the service addresses, the recorded protocol differs from `REPLACEMENT_PROTOCOL_VERSION` (2 on `main`), `bringup_bounds_unmeasured` if the bring-up bounds are

still placeholders, `tailnet_key_expiry` if, for the same kind of swap, the key the successor will use to join the private network has seven days or less left, and `automatic_migration_gate` if the candidate carries a schema migration. The comment on that last check says it must precede every side effect.

11.5.2. Birth, enrollment and qualification (steps 3 to 5)

The provider clones the successor from controller template 9101 and starts it. Chapter 12 covers templates and birth; what matters here is the newborn's posture. In the template only the enrollment unit is enabled. Every unit that would exercise factory authority is disabled, and so is the replacement driver.

The successor needs the candidate release and the controller's inputs, and it has no credentials yet. Enrollment solves that with a channel that works once. The driver asks a root-owned helper on the predecessor, over a local socket, to open the channel. The helper builds the release archive for the exact candidate and returns a key-only envelope. The driver claims the `controller_enrollment` effect and writes the envelope to a memory-backed path on the successor. The helper then listens for the successor's observed address only. The runbook gives the limits: at most three authenticated sessions, and revocation after five failed proofs or the recorded lifetime. The successor proves the key together with its fresh machine and SSH identity, fetches a manifest, the archive and the inputs, and acknowledges with the digests of what it received. The listener closes and the key is destroyed. If a later driver pass finds an enrollment that was claimed and never confirmed, it closes the helper and ends `needs_operator`. It never issues a second key.

The successor then installs the candidate and writes a qualification document, `yesod.controller-local-qualification.v2`. The runbook lists what it attests: both release links resolve to the candidate tree, the running `yesod` module lives under that tree, every restored input matches its manifest, enrollment is acknowledged and closed, every authority unit is inactive, no service address is present, and the local machine does not own the authority row. Only then is the successor `passive` and the intent `ready`.

Up to here the predecessor has kept serving and merging. The option `--stop-before-handover` on `yesod replacement drive` (commit `59395e720`, 2026-11-16) makes a pass return "stopped before handover" at this line, so that birth can be exercised without the rest.

11.5.3. The handover (steps 6 to 9)

`ControllerReplacementHandover.handover` in `replacement_handover.py` begins by requiring a closed enrollment (`require_enrollment_closed`) and refuses to run on the successor. It advances the intent to `pausing` and calls `acquire_replacement_stop_guard`.

That function takes an admissions pause in the swap's own name, with its own token, and in the same transaction counts what is in flight. Quiet means no refinery attempt in `claimed`, `preparing`, `gating`, `gate_green`, `pushing` or `deploying` (`_BLOCKING_ATTEMPT_STATES`) and no gate execution in `submitted`, `starting` or `running`. If the count is zero the guard row becomes `proved` and an immutable row is written to `replacement_quiet_gap_observations`. If

anything is in flight, the pause is released in the same transaction, the handover defers with `ControllerHandoverDeferred` and the pass ends with the intent still in `pausing`. If someone else already holds a pause, the guard records `admissions_paused_by_other` and the handover defers in the same way. The swap never proceeds under another actor's pause. A deferral does not wait long: `_handover_deferral_deadline` allows an intent to stay in `pausing` for its `claim_hold_seconds` (default 30, at most 300), and a later pass that finds the bound passed ends the intent in `needs_operator` at `handover_deferred_deadline`. The guard also returns a "disabled" proof unless the intent's `automation_enabled` column is true, and the column defaults to false.

One interlock follows from this and bears on the merge-triggered case. It is a static reading of the source on `main`; no test or run has exercised the path. The query in `_capture_quiet_gap` selects every attempt in `_BLOCKING_ATTEMPT_STATES`. It does not read `replacement_intent_attempts`, so it makes no exception for the attempts the intent itself covers. The attempt that triggered a swap stays in `deploying` for as long as its intent is not terminal, because `classify_status_for_settlement` reports a live intent as pending. As coded, then, a handover started while its own triggering attempt is still `deploying` cannot prove quiet: it defers, and the deferral bound above ends the intent in `needs_operator` at `handover_deferred_deadline`. The proof can succeed only after that attempt has left `deploying`, and with the intent still live the exit the refinery offers is the settle timeout to `merged_not_deployed` (Chapter 10). In that case step 15 would settle a later retry attempt for the same candidate, not the attempt that started the swap. A swap whose intent was submitted by hand has no attempt in `deploying` and is not affected. This reading was first made during the writing of this edition. On the evening of 2026-11-19, according to the mayor's board notes, two reviewers confirmed it independently and it was filed as FINDING #132 (see "The evening of 2026-11-19" below).

With the gap proved, the intent moves through `draining` to `fencing`. The driver asks the provider to hand over to the successor at the next lease generation, then performs the transfer that matters. `_move_leadership` calls `transfer_controller_authority`, whose docstring is the rule: "CAS-transfer ownership; expiry alone never authorizes takeover." The function checks that the caller owns the live pause at the expected revision, that the row still names the expected owner at the expected `revision` and `lease_generation`, and that every lane-leadership row is held by that owner at that lease generation. One `UPDATE` then rewrites the owner and increments both numbers, the lane rows follow, and a `transferred` event is appended. From this commit on, a lane write the predecessor attempts carries a stale lease generation and fails the leadership check (Chapter 8). The intent is `activating`.

The driver next calls the provider's `prepare_self_retire`, marks the predecessor target `stopped_retained` and the successor `serving`, and advances the intent to `settling`. The predecessor machine is still running and still holds the service addresses. Everything so far can be undone by machine.

11.5.4. The point of no return (step 10)

For an intent whose recorded contract sets `service_address_takeover`, the predecessor now gives up its addresses. The driver claims the `release_service_addresses` effect and calls the root helper. The runbook gives the order inside the helper: validate, rewrite the local birth

record to `retiring`, rename four activation markers with a `.retiring` suffix so that no factory unit will start on this host again, and only then take the addresses down.

The runbook names this line: "Address release is the irreversible automation boundary." A failure before the addresses go down restores the markers and the predecessor keeps serving. A failure once they are going down restores nothing. After the effect has been claimed, the code refuses to reactivate the predecessor, to reverse the compare-and-swap, or to roll back, whatever evidence state the effect is in.

11.5.5. After the boundary (steps 11 to 15)

The driver records the handover receipt and then claims and issues the predecessor's self-stop. For an intent that moves the service addresses, both places that can claim a self-stop first call `require_service_addresses_released`, which demands the release effect in `confirmed_applied` with an observation time no later than the receipt's retirement time. Self-stop is the predecessor's last act.

Between address release and the successor taking the addresses, nothing answers on them. The runbook calls this "an intentional relay and telemetry gap" and says to treat it as a visible outage of the swap, with admissions still paused.

The successor's own driver now passes its start condition, because the successor owns authority at the recorded lease generation. On each pass it renews that lease generation. In `_settle` it waits out the predecessor's shutdown allowance plus `DEFAULT_PREDECESSOR_STALE_SECONDS` (120), then requires the predecessor to be stale as judged from the database. A predecessor that still looks fresh, or whose liveness is unknown, ends the intent in `needs_operator` at `predecessor_staleness`. The successor makes no provider calls to find out; `successor_activation.py` has no provider transport and no token.

With staleness proved, `arm_successor_service_activation` writes a request file, and a root unit, `yesod-successor-activate`, retried every 30 seconds by its timer, does the takeover. It rechecks ownership of the authority row, calls the same `require_predecessor_effects` that handover uses, probes for a duplicate of each service address, brings the addresses up, repeats an egress check, restores the activation markers, rewrites the birth record from `isolated` to `activated`, and starts six units (`NON_PROMOTION_UNITS`). It waits for a scheduler tick from this successor before enabling promotion, then confirms the `successor_service_activation` effect. The shared predicate `successor_service_activated` requires three facts: the confirmed effect, the matching local receipt, and a tick after activation.

Only behind that predicate does `_settle` call `release_replacement_stop_guard` and `publish_deployment_receipt`. The receipt is a `yesod.replacement_deployment.v1` document and its publication moves the intent to `succeeded`. In the refinery, `CommandDeploymentStepper` reads the intent's status through `classify_status_for_settlement`; an applied receipt for the exact candidate settles the attempt as `succeeded`, and a `rolled_back` or `needs_operator` intent settles it as failed, which Chapter 10 describes as `merged_not_deployed`. For a merge-triggered swap this step is subject to the interlock described under the handover: as the code reads, the attempt that submitted the intent would have left `deploying` before the swap could finish, and the attempt settled here would be a retry.

11.6. The Invariants

Each rule below is enforced by the named symbol on `main`.

1. Authority moves by compare-and-swap, never by expiry. `transfer_controller_authority` requires the expected owner, revision and lease generation, matching lane leadership, and an owned pause (`_require_owned_admission_pause`). An expired lease gives no one the right to take over.
2. The swap owns its pause. `acquire_replacement_stop_guard` takes the pause, records ownership, observes the quiet gap and releases on failure in one `REPEATABLE READ` transaction. Its docstring says a failed proof cannot leak the pause and can never release another actor's pause.
3. No automatic swap carries a migration. `ReplacementMigrationCoordinator.complete_automatic` raises `AutomaticMigrationRefused` when `classify_expand_contract` reports `stop_required` or any pending migration id.
4. Unmeasured budgets block birth. While either bring-up bound equals its placeholder (`UNMEASURED_SUCCESSOR_BRINGUP_SECONDS`, `UNMEASURED_ENROLLMENT_TTL_SECONDS`, both 1800), `bringup_bounds_measured` is false and the driver ends the intent before any provider call. `ReplacementSuccessorContract` also rejects an activation budget that is not more than 30 seconds below the authority lease.
5. Enrollment is single-use. One key per intent, with the session and failure limits above; `require_enrollment_closed` guards every entry to handover.
6. A successor never completes its own takeover through the provider. `handover` raises `ControllerPredecessorStopPending` when the executor is the successor, and activation is database-proven.
7. Self-stop is last. `require_service_addresses_released` precedes both self-stop claim sites (commit `50ef40fc0`, 2026-10-27).
8. No rollback after address release. `ControllerReplacementHandover.rollback` refuses with "automatic rollback is forbidden after service address release was claimed" if the release effect exists at all, and ends the intent in `needs_operator` at step `rollback_after_release`.
9. The driver starts only where it should. The service's `ExecCondition` (`replacement_driver_condition`) lets a retiring controller run the driver only while a live intent names it as predecessor, and lets an isolated successor run it only inside its bring-up window or, from `fencing` on, only if it owns authority at the recorded lease generation. Any read error skips the driver.

> **Design Trade-off — migrations left the transaction.** The August promotion transaction ran the schema migration inside the deployment, with the fleet held (Chapter 10 tells that history). The automatic swap refuses any candidate with a pending migration (commit `8001389bc`, 2026-10-27), and the error text says where such a candidate goes: "ship via a hand window". This is a retreat in automation. It was chosen because a swap that is safe to abandon at any step before the boundary cannot also contain a schema change that the predecessor's code may not tolerate. The cost is that every migration waits for an operator window (Chapter 18, Signed Human Authority), and according to the agents' working notes, notes carrying migrations are held until one is scheduled.

11.7. Adopting a Hand-Born Controller

The driver expects the serving controller to have come from an earlier intent. The first time, there is no earlier intent: the serving controller was built by hand and no authority row exists. `yesod replacement adopt` bridges that once.

Adoption is the most tightly restricted command in the framework. `adopt_replacement_controller` in `replacement_adopt.py` requires root (`_require_root`), requires the actor to be `stephen` (`REQUIRED_ACTOR`), and requires the live admissions pause to be held by that actor at the stated revision with the matching token (`_owned_pause_conditions`). The scope must have no replacement intents and the refinery must be running in `ACTIVE` mode. The operator supplies a small reviewed document, `yesod.replacement_adoption.v1`, naming the controller's VM, hostname, machine id, network card address and serving release; every field must match what the command observes.

The command initializes the authority row and then waits for evidence from the running refinery: a later tick that renews the new lease generation and shows every configured lane held at it. The wait is capped by `MAX_PROOF_TIMEOUT_SECONDS` (300). It does not release the pause; the operator resumes admissions separately after reading the proof. Exit code 1 (`EXIT_ERROR`) is an error, and the operator must still establish that it happened before initialization. Exit code 3 (`EXIT_NEEDS_OPERATOR`) means authority was initialized and the proof is missing; the pause stays held and nothing is cleaned up.

> **Factory Floor — four refusals in one day (2026-11-10).** The agents' working notes record that adopting the serving controller was refused four times on 2026-11-10 before it succeeded at 23:20Z. Each refusal was a precondition that could have been read in the code beforehand. The pause had been taken by the refinery clerk, and adoption accepts only one owned by `stephen`. The network inventory raised an error on a tunnel interface that has no hardware address (fixed by `ad55cf61d`: `_local_nic_macs` now considers only Ethernet interfaces). The provider token lacked an audit permission. The proof wait defaulted to less than the refinery's real renewal period, and the wrapper script then deleted the adoption document although the database had already been changed (fixed by `eef271b34`, which set the fixed 300-second budget; the runbook's `adopt_with_retention` wrapper never deletes the document). The lesson the notes draw is to walk the called code to its first write and past it before asking a human to run it.

11.8. The Dry Run

The four refusals produced the next mechanism. Commits `bcedcc1cb` and `93e95c439` (2026-11-12) added a read-only preflight to both commands: `yesod replacement adopt --dry-run` and `yesod replacement drive --dry-run`.

A preflight differs from a real run in two ways. It writes nothing: the runbook says it neither initializes nor renews authority, acquires no leadership, mutates no intent and makes no provider mutation. And it does not stop at the first failure. `PreflightCollector` in `replacement_preflight.py` gathers every failed precondition into one report, and a check whose inputs are unavailable says so.

```
sudo -u yesod-control yesod replacement drive --dry-run
```

The runbook sets four conditions for treating the result as green: the command exits 0, `document_kind` is exactly `yesod.replacement_preflight.v1`, the report's root `ok` is true, and every entry in `checks` has `ok` true. The report also measures how often the refinery really renews its authority (waiting at most `MAX_CADENCE_SECONDS`, 90) and warns if a proof budget is shorter than twice that period.

A green report is a prerequisite for requesting a grant in the same window (Chapter 18). It authorizes nothing, and the real commands recheck everything.

11.9. When a Swap Fails

What happens depends on where the failure falls relative to two lines: the first provider handover and the address release.

11.9.1. Before the handover (steps 1 to 5)

The predecessor is untouched and the intent ends `needs_operator`. Where a birth receipt exists, `_terminalize_preparation_failure` in `replacement_worker.py` asks the provider to retire the failed successor, marks its target `failed`, and records the obligation not to reuse it; disposing of the machine is left to an operator. A later intent may follow such a failure when `is_preauthority_birth_failure` in `replacement_live_state.py` can prove that the terminal birth failure left predecessor authority untouched. That proof was added in mid-November and then extended to failures the driver records before any provider birth.

11.9.2. In the handover, before the provider handover

The exception path in `handover` releases the swap's pause, because the successor was never activated, and ends `needs_operator`.

11.9.3. After the provider handover, before address release

The pause is kept. If the predecessor had already been prepared to retire, `_recover_failed_handover` reverses the work: it reactivates the retained predecessor, hands over to it at the next lease generation, moves authority back with a second compare-and-swap, retires the successor, marks the targets accordingly, releases the pause and records the intent as `rolled_back`. If that recovery itself fails, or the failure fell between the provider handover and the preparation, the intent ends `needs_operator` with the pause held. A separate method, `ControllerReplacementHandover.rollback`, performs the same reversal on request from the successor in `activating`, `rolling_runners` or `settling`; at this snapshot nothing on `main` calls it outside the test suite, so the driver has no path that reaches it.

11.9.4. After address release

Nothing is automatic. The intent ends `needs_operator`, admissions stay paused, and the pause rule recorded for a stale predecessor with an unactivated successor is `leadership_confirmed_successor_not_activated_hold`. The runbook's manual procedure is to keep both controllers fenced, read the release and activation effects and the local receipts, prove which host holds each service address, repair by hand, choose one controller to reactivate, and resume the pause as a separate step. It adds an ordering rule for a manual return to the predecessor: make the intent terminal first, then boot the predecessor. A predecessor booted while its intent is still `settling` can be powered off again by the next driver pass.

11.10. Proven Where

> **Proven Where — as of 2026-11-19.** Designed and on `main`: the whole sequence above, from submission to receipt. Run in production: adoption of the serving controller (2026-11-10) and a green dry run (2026-11-12), both according to the agents' working notes. Entered in production and not completed: birth and enrollment of a driver-born successor, in three attempts that each ended needing an operator. Never run in production: the handover (pause, authority compare-and-swap, address release, self-stop, activation), automatic rollback, and a merge-triggered swap with no operator. No replacement intent has reached `succeeded` in production. Open: a merge-triggered handover cannot prove quiet while its own triggering attempt is still `deploying` (FINDING #132, confirmed by review on 2026-11-19 and never exercised), so the merge-triggered swap has not run anywhere, in production or in rehearsal. As of the evening of 2026-11-19 no fourth attempt has been made.

Capability	On `main`	Production
Controller replaced by hand	runbooks	yes, since 2026-09-16
Authority row and CAS transfer	yes	initialized only (11-10)
Adoption	yes	yes, 2026-11-10
Dry-run preflight	yes	green 2026-11-12
Driver-born successor (steps 3-5)	yes	three attempts, none completed
Handover (steps 6-14)	yes	never run
Automatic rollback	yes	never run
Merge-triggered swap, no operator	wired; blocked by #132	never run anywhere
Swap carrying a migration	refused	by hand window only
Runner replacement	yes	never run

Rehearsal on disposable machines sits between those two columns. Hand-submitted swaps have been rehearsed on disposable machines using reference code. The first birth of the built

hardening code on a rehearsal box was on 2026-11-19 and is what exposed finding #136. A merge-triggered swap has never run anywhere. This is as recorded in the agents' working notes and was not otherwise verified for this edition.

11.10.1. The controllers so far

On 2026-09-16 a fresh VM, VM175, replaced the original controller VM, which was left fenced and powered off with its disks retained (`docs/runbooks/controller-g1-handoff-2026-09-16.md`). Later hand-run controller host generations followed; according to the working notes, g4 served from 2026-10-26 and g5 from 2026-11-03. None of these used the driver.

11.10.2. The attempts

The replacement framework landed as code on 2026-10-13 and 10-14; the driver, the adoption command and the successor protocol followed between 10-26 and 10-30. After the adoption and the green dry run came the production attempts, as the agents' working notes record them. In mid-November 2026 three replacement attempts were submitted by hand in production. Each stopped during the successor's birth or enrollment, before any transfer of authority, on a different precondition that only a real run exposed. Each ended `needs_operator`, the serving controller kept serving throughout, and each was followed by a root cause and a fix before the next request. The intents were submitted under signed grants (Chapter 18), not by a merge.

None of the three reached step 6, and none transferred authority. The serving controller, VM179, kept serving throughout. The handover has no production history.

> **Factory Floor — green dry run, red production (mid-November 2026).** The last of the three attempts followed a day of reviews, a written pre-mortem and a green dry run. It still stopped, and a rehearsal of the same bring-up on a scratch machine afterwards found further blockers. The working notes record that the same rehearsal had been declined earlier to save time. The dry run had been green because a dry run checks preconditions, and these failures were in steps that had never executed anywhere. The fixes were committed under one note (ys-yes-gn4b) and are on `main` in the candidate commit `16606a42`, which this edition takes as its snapshot (stamped 2026-11-17 19:16Z, a time derived from its source commit, not the time of landing; the working notes place the merge window on 2026-11-18). The rule recorded afterwards: for a path that has never run end to end, a rehearsal on a disposable machine comes before any request to run it in production.

11.10.3. The evening of 2026-11-19

What follows is as of the evening of 2026-11-19 and comes from the mayor's board notes for that day. It is in the working-notes tier; where a claim was checked against `main` the text says so.

Ahead of a fourth attempt the factory built hardening work the agents call H1, by its normal route: a prose note, nineteen child tasks, a dispatched worker on the build seat. H1 is not on `main`. On 2026-11-19 it passed its full Linux test union, 4,591 passed and 0 failed, measured on a disposable Linux test container. Then, in one evening of pre-flight and rehearsal before any production step, seven findings were raised. None touched production.

- **#130, the fourth submit would have been refused at its first command.** The proof that clears earlier failed intent generations compared each generation's recorded set of applied migrations with the new candidate's migration list. H1 adds a migration, so the three earlier failures could never be cleared. It was found by a read-only run of the new code against production's records. Every test box had used a single migration ladder, so no test could see it.
- **#131, a service the predecessor never ran.** A born successor would have started a release-promotion service that the serving controller has never run. Stephen's ruling, as recorded: keep it off and mirror the predecessor.
- **#132, a merge-triggered swap deadlocks as coded.** This is the interlock described under "The handover". The triggering merge's attempt waits in `deploying` for the swap, and the quiet-gap proof (`_capture_quiet_gap` in `replacement_guard.py`) requires zero attempts in `deploying`, with no exemption; that much is verified on `main`. Every swap so far, in production and in every rehearsal, had been submitted by hand, so the path was never run. The reading was first made by a documentation pass, the fact-check of this chapter, and then confirmed independently by two reviewers. Said plainly: a documentation pass found a blocker on the final step of the factory's first goal.
- **A clock mismatch beside it.** The controller's refinery lane gives a deployment 30 minutes to settle (`settle_seconds: 1800` in the reviewed controller document on `main`), while a swap's deadline in the notes is two hours. A swap that ran long would see its triggering attempt time out to `merged_not_deployed` first.
- **#133 (deferred).** Two merges inside one swap can make the older attempt submit its candidate again. Mitigated for now by admitting one merge at a time.
- **#134 (deferred).** A successor that is born and enrolled and then fails bring-up is left running, and no recovery verb accepts its state. A reviewed manual exit is documented instead.
- **#135 (deferred, scope still being traced).** The driver does not serialize passes on one intent. A pass that dies between a successful VM clone and the record of it can lead the next pass to record "not applied" over a running VM.
- **#136, an enrollment that could never finish.** H1's own enrollment step required three services to be active. The birth fence deliberately holds those services down until activation: on `main`, `build-controller-template.sh` gives each authority unit a drop-in with `ConditionPathExists=/etc/yesod/controller-activated` and disables it. So no successor could have finished enrolling. The H1 check is not on `main` and is described here as reported.

> **Factory Floor — the stand-in said "active" (2026-11-19).** Finding #136 passed code review and the full test union of 4,591 tests. It passed because the unit-test stand-in for the service manager answered "active" for every unit, including the ones the fence exists to keep down. It was found on the first real birth of the built code on a rehearsal box. The notes trace the cause to the specification: the requirement said what to prove and did not say in which phase. Two lessons were recorded. A proof requirement names its phase. A test stand-in must model the constraint the code is meant to respect.

Two process lessons from the same evening are recorded in the mayor's words. Approvals for an agent-written patch were requested twice on a partial test run (touched files only, not on Linux), and both requests had to be replaced; the rule now reads "no approval request until the full Linux union is green on the exact tree". And reviewers who had read every byte of a change had not run the existing tests that depended on the changed code.

The evening is the strongest case in this book for rehearsing before asking. Two of the findings (#130 and #136) would each have stopped a hand-submitted fourth attempt in production, one at its first command and one at enrollment, and #132 would have stopped the merge-triggered swap that is the goal. Each was found for the price of a read-only query, a disposable machine, or a careful reading.

11.10.4. Where it stands

As of the evening of 2026-11-19 the hardening patch is on its fifth revision and still in its test gate. No fourth attempt has been made. A database migration is part of H1, and by invariant 3 it must go through a hand window. The automatic, merge-triggered swap has never run anywhere, including in rehearsal.

The goal the factory's plan calls G1, "deploy on merge", has a precise test in the working notes: one controller swap triggered by a merge, with no hand-run step and no operator fix during the swap, ending in a published receipt. That has not happened, and finding #132 means that it could not have happened with the code as it stands on `main`.

Source Notes

Claim or section	Tag	Anchor
Why a swap	[deploy], [notes]	`docs/runbooks/routine-deployment.md`; agents' working notes, 2026-10-13
Durable state, phases, target states	[code]	migration `0117_replacement_promotion_state_ys_yes_mabj.py`; `replacement_contract.py` (`_PHASE_TRANSITIONS`); `replacement_store.py`

Claim or section	Tag	Anchor
Authority row	[code]	migrations `0097_controller_authority_ys_yes_ectq.py`, `0118_controller_leadership_fence_ys_yes_mabj.py`; `controller_authority.py` (`transfer_controller_authority`, `_require_owned_admission_pause`)
Trigger and coalescing	[code]	`routine_deployment.py` (`apply_candidate`); `replacement_cli.py` (`submit_replacement_intent`, `EXIT_NONTERMINAL`); `ReplacementStore.coalesce_intent`
Driver cadence, disabled by default	[code]	`ops/yesod-dispatch/systemd/yesod-replacement-driver.timer`; `docs/runbooks/replacement-driver.md`; `ReplacementDriver.drive`
Timer kept disabled on the live controller	[notes]	agents' working notes, 2026-11-11
Steps 1-2	[code]	`replacement_driver.py` (typed `failed_step` values); `replacement_migration.py`
Steps 3-5	[code]	`docs/runbooks/replacement-driver.md` (enrollment, qualification); `docs/runbooks/controller-birth.md` (template 9101); `replacement_enrollment.py`
Steps 6-11	[code]	`replacement_handover.py` (`ControllerReplacementHandover.handover`); `replacement_guard.py`; `replacement_host_local.py`
Quiet gap and the triggering attempt	[code], [inferred]	`replacement_guard.py` (`_capture_quiet_gap`); `replacement_driver.py` (`_handover_deferral_deadline`); `replacement_cli.py` (`classify_status_for_settlement`). Read statically; never run
Steps 12-15	[code]	`replacement_driver.py` (`_settle`, `DEFAULT_PREDECESSOR_STALE_SECONDS`); `successor_activation.py`; `yesod-successor-activate.timer`; `refinery_deployment.py` (`CommandDeploymentStepper`)
Invariants	[code]	symbols named in each item; commits `8001389bc`, `50ef40fc0`

Claim or section	Tag	Anchor
Adoption mechanism	[code]	`replacement_adopt.py`; `docs/runbooks/replacement-driver.md`
Four refusals; adoption at 23:20Z	[notes]	agents' working notes, 2026-11-10; commits `ad55cf61d`, `eef271b34`
Dry run	[code]	`replacement_preflight.py`; commits `bcedcc1cb`, `93e95c439`; runbook
Failure handling	[code]	`replacement_handover.py` (`_recover_failed_handover`, `rollback`); `replacement_worker.py`; `replacement_live_state.py`; runbook
VM175 handoff	[deploy]	`docs/runbooks/controller-g1-handoff-2026-09-16.md`
g4, g5 dates; the three hand-submitted attempts; rehearsal; H1; held migration notes; goal G1 test	[notes]	agents' working notes, 2026-10-26 to 2026-11-19
Evening of 2026-11-19, findings #130 to #136	[notes], [code]	mayor's board notes, 2026-11-19; `replacement_guard.py` (`_BLOCKING_ATTEMPT_STATES`, `_capture_quiet_gap`); `ops/yesod-birth/build-controller-template.sh`; `controller-reviewed-document.json`

Not verified:

- The three production attempts of mid-November are told in general terms from the agents' working notes; their individual causes and dates are left out of this edition.
- Everything under "The evening of 2026-11-19" except the two points marked as checked on `main` (the blocking states of the quiet-gap proof; the birth fence in `build-controller-template.sh`). The test count of 4,591, the findings #130 to #136 and the state of the patch are as reported in the mayor's board notes.
- What has been rehearsed on disposable machines is as the working notes record it.
- The interlock between the quiet-gap proof and a triggering attempt in `deploying` is read from source only (`_capture_quiet_gap` has no exemption for the intent's own attempts; a deferral in `pausing` is bounded by `claim_hold_seconds`). No test or run was found that exercises a merge-triggered handover end to end, and whether an operator path exists to move the attempt out of `deploying` sooner was not traced.
- That `ControllerReplacementHandover.rollback` has no caller outside tests rests on a symbol-reference index and a text search of `yesod/src` and `ops` at `16606a42`; a dynamic call would not be found that way.
- The exact driver pass on which an intent moves `planned` → `birthing` → `qualified` → `ready`. The table follows the research brief and the runbook.

- The contents of H1 and of the pending migration, and whether the off-production handover rehearsal completed.
- Production values in the binding (measured bounds, lease and activation budgets) and whether the live controller document carries the deployment block. No host was inspected.
- The interaction between the deployment settle time in the reviewed controller document and the length of a full swap was not traced.

12 Born, Not Repaired

Chapter 11 (Deploy Is a Host Swap) describes how authority moves from one controller to its replacement. This chapter describes where the replacement comes from. Every factory machine except the hypervisor and the data stores is meant to be disposable: a runner, a gate VM or a controller is created from a versioned recipe, proves who it is, receives its credentials last, and leaves behind a record that later steps refuse to proceed without. Yesod calls that sequence a **birth**.

The reader will learn the seven steps of a birth, follow one gate VM through them, see how runner and controller births differ, and learn where credentials, backups and blobs fit. The chapter also says plainly which parts of this have run in production and which have not, and what the topology figure at the end rests on.

12.1. Why Rebirth

Until early September 2026 the factory's machines were pets. The agents' working notes for 2026-08-30 record runners as long-lived by settled interim policy, because each one needed a human to log in to the model providers. A machine that is hard to create gets repaired in place, and a machine repaired in place slowly stops matching whatever created it. The notes record the same model-configuration drift being corrected by hand on runners twice.

Three things changed that. An inference gateway VM took over the provider logins and began issuing a key per runner seat, so a runner no longer needed a human at birth (first proven on 2026-08-31, per the working notes). The scripts that created machines moved into Git. And Stephen stated the policy. The working notes record him on 2026-09-22: "remember, 'cattle, not pets'. I would prefer rebirthing the runners rather than changing them in place", and on 2026-09-24: "we should be in a position where we do not hesitate to redeploy".

The case for rebirth over repair has three parts, each visible in the code. A hand-changed host drifts from its recipe, and nothing detects the drift. A born host carries attestations that an in-place change cannot renew: the gate template bakes the forced-command program whose digest the fleet inventory pins, so reinstalling that program on a live gate breaks the pin, and the supported answer is to rebuild the template and birth a new set. And each birth exercises the recipe, so a recipe that has rotted is found on a machine nobody depends on yet.

Factory Floor — The recipe lived on one machine. Until 2026-09-09 the scripts that created every runner, gate VM and test database existed only in a directory on the hypervisor sefer, unversioned, with change history kept as `.pre-<change>-<timestamp>` copies beside the originals. Commit `5f8582b8a` moved them into the repository as `ops/yesod-birth/`. Its message gives the reason: "a host generation is meant to denote the birth recipe it was built from, which is meaningless while the recipe is unversioned". It also records that stale copies of the gate harness were why per-shard gate diagnostics came back empty. `ops/yesod-birth/README.md` now states the rule: the repository is the source of truth, the directory on sefer is a deployment copy derived from a reviewed commit, and a local repair made during an incident is back-ported to Git before the next deployment.

12.2. The Seven Steps

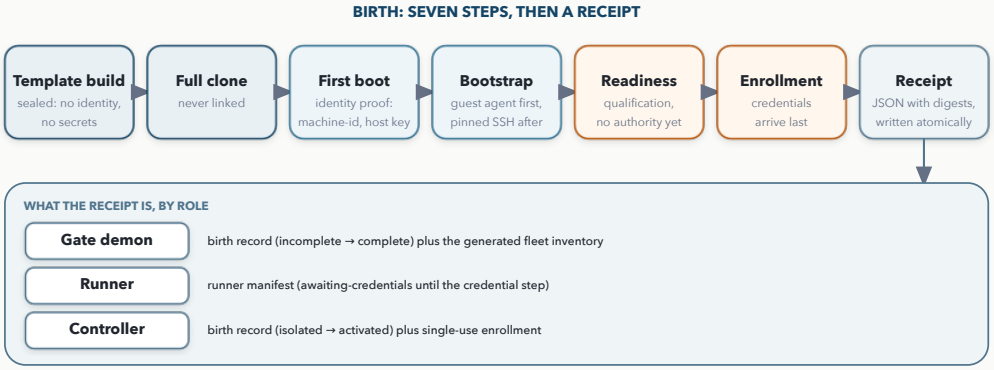

Figure 12.1.: A birth runs from a sealed template through clone, identity proof, bootstrap, readiness and enrollment to a receipt, and credentials arrive only after identity is proven.

Every role follows the same order.

1. **Template build.** A script builds a sealed Proxmox VM template from a reviewed Git commit. The template holds software and no identity or secrets.
2. **Clone.** The newborn is a full clone, never a linked one, so it does not depend on the template afterward.
3. **First boot and identity proof.** The newborn must show a fresh machine id (`assert_fresh_machine_id`) and an SSH host key that is the same on disk and on the wire (`assert_hostkey_wire_disk`).
4. **Bootstrap.** Role software and configuration are installed through the hypervisor's guest agent first. SSH is used only after identity is proven, and only with a pinned key.
5. **Readiness.** A check that the newborn can do its job, made without giving it authority.
6. **Enrollment.** Credentials and membership arrive from outside, last.

7. **Receipt.** A JSON record with digests, written atomically. Later steps read it and refuse to proceed without it.

The order matters most at steps 3 and 6. A machine that has not proven its identity receives nothing secret, and a machine that has not passed readiness receives no authority. A newborn runner is written with `max_concurrent: 0`. A newborn controller has every unit in `AUTHORITY_UNITS` fenced.

Births on `main` default to three templates (the README also lists older gate templates kept only for deliberate rollback). The shared base (VMID 9100, `yesod-node-template`) is never modified and is the source for runners, hand-born controllers and the other two templates. The gate template (VMID 9114, `yesod-gate-template-dg6-3`, the value of `DEFAULT_GATE_TEMPLATE_VMID`) is built by `build-gate-template-dg5.sh`, which refuses to replace an existing VMID and refuses the shared base as a target. The controller template (VMID 9101, `yesod-controller-template`) is built by `build-controller-template.sh`, which will not seal the template until the machine id is empty, host keys and tailnet state are absent, and every controller-input path is empty.

12.2.1. What "generation" means

Names carry a generation, and the word is easy to misread. For runners (`rgN` in the README, `g<N>` in the VM name) and gate VMs (`dgN`), the README is explicit: a generation "denotes the versioned birth recipe used to create it, not how many times a fleet member has been replaced". Rebirthing from the same recipe keeps the generation and takes a new random suffix. A change to the recipe earns a new number. A gate VM is named `yesod-gate-dg<N>-<demon>-<suffix>`, with the demon drawn from a pool of 115 names (commit `6b488114f`) by `pick-name.sh`, which refuses collisions. A runner is `yesod-runner-g<N>-<name>-<4 hex>`.

For the gate fleet the generations that matter on `main` are: g1, the original hand-built fleet; dg5, the first fleet born from the recipe; and dg6, whose births are bound to a prepared, role-keyed inventory from which time and egress policy are rendered (commit `c69bc2148`). Generations 2 to 4 survive only as a legacy branch in `birth-gate.sh`. Chapter 9 (The Deterministic Gate) describes what the fleet does.

Controllers are named `yesod-controller-g<N>-<4 hex>`, and `Birth.validate` in `birth-controller.py` requires the number in the name to equal `--generation`. The README's recipe-label statement covers runners and gate VMs only. For controllers the number has advanced with each hand-run replacement: the runbook names the first as g1 on 2026-09-16, and the working notes record g4 serving from 2026-10-26 and g5 from 2026-11-03. This controller host generation is one of the four kinds of generation that Chapter 11 separates. It is a different counter from the intent generation and the lease generation defined there, and from the count of three production attempts at a driver-born successor. The host numbering and the attempt count have not been reconciled against a single source for this edition.

12.3. One Birth, Traced: a Gate VM

A gate VM, called a **demon**, is the best example because its birth uses every part of the system. The trace below follows one dg6 demon for the role `sefer-shard-0` as `birth-gate.sh` is written on `main`, and says where the dg5 path differs. It is a reading of the script, not a log of a production run. All commands run as root on sefer and all of them change state.

Prepare the inventory first. Before any demon exists, the operator runs the `prepare` phase of `generate-gate-fleet-inventory.py` (`prepare_inventory`). The prepared document is keyed on the nine roles in `ROLE_NAMES` (`sefer-shard-0` through `sefer-shard-7` and `sefer-safety-island`). In the README's words it contains "no VM name, machine identity, host key, or database capability selected from a newborn". Its SHA-256 digest is the thing every birth in the set will be bound to.

Birth the demon.

```
birth-gate.sh <demon> 6 <suffix> \
  --binding-role sefer-shard-0 \
  --rendered-inventory /path/to/sefer-gate-fleet.prepared.json
```

`resolve_gate_profile` reads `gate-profiles.json` for the role: a shard gets 8 cores and 8192 MiB, the safety island 2 cores and 2048 MiB. For generation 6 the script calls `build_dg6_birth_contract` in the inventory generator, which validates the prepared inventory and binds the requested name, suffix and role to its digest. `assert_template_identity` checks that the template VMID and template name match as a pair. On the dg5 path, `prepare-gate-readiness.py check` also renders the network packet and verifies the recipe tree. All of this happens before anything is cloned.

`clone_and_boot` then makes the full clone and starts it. `wait_agent` waits for the guest agent, and since commit `de24c10a9` (2026-09-17 UTC) it requires a real guest command to succeed, because an agent ping can answer a moment before commands are served and the first source build of a template died on exactly that race. `install_host_key_persistence` writes the cloud-init drop-in that keeps host keys across reboots.

Prove identity. `assert_fresh_machine_id` rejects a machine id inherited from the template lineage. `assert_hostkey_wire_disk` reads the host key from disk through the guest agent and compares it with the key the network presents. The script logs `identity OK` only when both pass.

Write the record as incomplete. `persist_incomplete_birth_record` writes the birth record with `provisioning_state: incomplete`. From this point a crash leaves evidence of a half-born machine and never a record that looks finished.

Reserve the address and prove it (dg5 path). In `birth-gate.sh` this step runs only for generation 5. `reserve_dg5_lease` reserves the DHCP lease the newborn was observed to hold, then queries the lease and requires that it match. The log line is "observed DHCP reservation in effect". An accepted request is not treated as proof. `birth-runner.sh` does the same for generation 4 runners in `reserve_g4_lease`.

Readiness. The script installs logging for both generations. On the dg5 path it then activates the network, time and egress policy and records a release attestation (`prepare-gate-`

`readiness.py install`). The README says generation 6 "keeps its separate inventory-driven activation": the same policy is rendered from the activated inventory by `render-gate-network.py` after the bind step below. Two refusals are written into the README: a logging failure prevents the final record, and, for the dg5 readiness receipt, "a receipt without the NTP-source proof never completes a birth".

Complete the record. `persist_complete_birth_record` rewrites the record with `provisioning_state: complete` and `readiness_state: job-ready`. The schema (`gate-birth-record.schema.json`) fixes `record_type` as `yesod.gate.birth` and `role` as `gate-node (zero-credential)`, and requires the machine-id digest, the host key, `binding_roles`, `paired_database_capabilities` and `rendered_inventory_sha256`, which for a dg6 birth is the digest of the prepared inventory from the first step (a dg5 record carries a null there). The authoritative copy stays on sefer. A mirror to the NAS is best-effort.

Birth the paired database. Each demon runs tests against its own disposable PostgreSQL, a container born separately by `ops/gate/testdb-birth.sh` with its own record. The pairing is by name: the fleet preflight derives the expected database from the demon's name (`_paired_database_capability` in `preflight-sefer-gate-fleet.py`) and refuses any other. The working notes for 2026-09-18 record why this check earns its place. Nine successor demons were born pointing at their predecessors' databases, every in-guest check passed, and only the fleet preflight refused. It stopped a combined window that had already been opened. The lesson the notes draw is to run the fleet preflight on successors before spending a pause window.

Bind and activate. A birth does not put a demon into service. `manage-gate-fleet-set.py regenerate` (`regenerate_candidate`) builds the bindings, the activated inventory and the `known_hosts` file from the birth records. The `activate` phase of `generate-gate-fleet-inventory.py` (`activate_inventory`) fails closed on an omitted role, a changed record digest, an incomplete birth, a wrong generation, or a record born from a different prepared digest. The README calls binding "a reviewed operator choice, not a birth side effect" and forbids selecting a record by creation time, filesystem order, a `current` symlink "or whichever record appears newest". Per `docs/runbooks/sefer-dg6-fleet-sets.md`, while the new set is born and qualified in shadow the inventory carries it as a `candidate_generation` beside the active one, and activation is one generated transition that appends the old generation to `retired_generations`.

Deliver the credential. Described in the credentials section below: the catalog row is seeded by `reconcile-gate-fleet-credentials.py`, and the controller delivers and probes.

A whole fleet is born with `manage-gate-fleet-set.py birth-set`. `build_birth_set_plan` validates every name, VMID, role and template digest before the first clone, and `execute_birth_set` births one pair at a time and stops at the first failure. A set is between 8 and 16 pairs (`MIN_SET_SIZE`, `MAX_SET_SIZE`).

> **Factory Floor — An inventory that went stale within hours.** The checked-in file `ops/yesod-dispatch/config/sefer-gate-fleet.json` deliberately declares `status: not_an_activated_inventory`. The reason is an incident the agents' working notes date to 2026-09-11. The gate inventory was then written by hand and keyed on demon hostnames, with 54 name-bearing tokens. Within hours of a rebirth eight of nine names no longer matched, and one host key had changed after a reboot underneath a correctly entered value. Commit `339dac3c0` (2026-09-11) introduced the two-phase generator. The general lesson: key an inventory on what stays the same (the role), and generate everything that changes (names, keys, addresses) from the receipts of the machines themselves.

12.4. Runner Birth

A runner is born in two scripts with a deliberate gap between them.

`birth-runner.sh` checks its inputs before cloning (logging settings, the offline PostgreSQL package bundle and its manifest), then clones the shared base, proves identity with the same two assertions, reserves the lease (generation 4), provisions a local PostgreSQL, and publishes the runner's pins to the rollout inventory. It writes a `runner.yaml` that can only heartbeat: `max_concurrent: 0`, planner off, no database connection string. It ends by writing `RUNNER_MANIFEST.json` with `provisioning_state: "awaiting-credentials"`. The script's header lists what it never does: inject credentials, join the tailnet, or enable the dispatch daemon.

`credential-runner.sh` then runs numbered steps from 0/8 to 8/8: reachability, tailnet join, a per-seat inference-gateway key, a GitHub deploy keypair generated on the runner (the step label reads "private key never leaves the runner"), rendered environment files, a fresh clone at the accepted release, the blob-store mount, the release deploy, and the final `runner.yaml`. One act stays manual by design. The script's closing instructions say to enable the dispatch daemon "ONLY with planner authorization".

Retirement is the mirror image. `retire-runner.sh` describes itself as "fenced cattle retirement on Sefer (no destroy)": it refuses while a build is running, disables dispatch, releases the DHCP reservation with a receipt and revokes the seat's gateway key. `harvest.sh` then reads the stopped runner's disk read-only into a NAS archive. The README's instruction is "Harvest transcripts before destroying a host."

12.5. Controller Birth

The controller is born by two paths, both documented in `docs/runbooks/controller-birth.md`, which opens: "This replaces the VM, not just `/srv/yesod/current`."

The hand path is a sequence of scripts run by an operator. `birth-controller.py` clones the shared base, verifies the newborn, leaves it not starting at boot and fences every authority unit. `controller-bootstrap.py` installs runtimes, an immutable release and units. `controller-inputs.py` moves the controller's inputs from predecessor to newborn by a direct pipe; the

runbook says never to send the export to a terminal or an agent tool result. `controller-readiness.py` proves readiness, and `controller-cutover.py` pauses, fences and cuts over. The runbook is firm that a failed birth "is not permission to clone another VM": `--resume-isolated` accepts only the same unfinished intent and the same VM.

The provider path is the one the automatic replacement driver uses. A clone of the sealed controller template receives a small enrollment envelope through the guest agent and enrolls itself in two stages. Chapter 11 describes the enrollment channel, the successor's qualification and how far the production attempts got.

Both paths leave a chain of receipts defined in `controller-receipts.py`: `yesod.controller_rebirth.inp` `.birth`, `.readiness` and `.cutover`. `_reject_secret_values` refuses to write a receipt that appears to carry a secret. Per the runbook, the newborn's local record moves from `isolated` to `activated`, and the file named by `ACTIVATION_FILE` "is the last gated marker".

> **Proven Where — birth, as of 2026-11-19.** Gate birth has run in production: commit `2584d2844` records the active configuration as the dg5 fleet alone from 2026-09-16. Runner birth has run in production for the generation 4 seats listed in `routine-deployment-g4.json`. Hand controller birth has run in production: VM175 replaced the original controller VM on 2026-09-16 (runbook on `main`), and the working notes record later hand-born controllers through g5. Provider-born controller successors are on `main` and were rehearsed on disposable machines; in production a successor has been cloned but none has completed enrollment (Chapter 11). Measured birth timing, which the replacement design requires, is still a hold on `main`: `replacement-birth-baseline.md` records storage above the 85% projected-use ceiling and says not to copy an earlier gate birth duration.

12.6. Credentials Arrive Last

One rule covers all three roles: templates hold no secrets, and credentials arrive after identity is proven, through a channel that does not show the values to an operator or an agent.

Gates. A demon is designed to hold one secret, the credential for its own disposable test database, and to hold it only until the next reboot. The envelope lives on a tmpfs directory that, in the words of `sefer-gate-credential-lifecycle.md`, "must be empty at the start of every boot". The authoritative copy is an encrypted catalog on the controller (table `sefer_gate_credentials` for the original hand-built g1 gate fleet and `sefer_birth_credentials` for born generations, each with an audit log table beside it). The module docstring of `sefer_gate_credentials.py` explains the separation: "Gate DSNs are deliberately not fleet secrets: runner environment refresh must never distribute them."

Delivery works by observation. At boot, `yesod-sefer-gate-credentials.service` runs before the SSH daemon and creates empty state. The controller's `SeferCredentialSynchronizer` asks each demon for a names-only status over a dedicated `gate-credential` account, sees a new boot identity, sends the envelope only when the status is stale or unready, and requires the demon's materializer to pass a live database probe. The runbook tells operators not to copy an envelope by hand, because doing so bypasses the catalog, the boot identity check, the probe and the audit trail.

The catalog exists because of 2026-09-09. Among the eight silent failures the working notes record for that day (Chapter 13, Seeing the Whole System), gate database credentials kept on tmpfs had been destroyed by a reboot with nothing to put them back.

The mechanism is baked into the template; the secret is not. Since commit `1ba5312ab` (2026-09-15 UTC) the gate template builder bakes in the materializer, the boot unit, the forced commands and four public keys, so a clone needs no credential install step after birth. The operator commands are `yesod fleet gate-credential set`, `rotate`, `list`, `status`, `delete`, `refresh` and `audit-log`. Values are read from a protected file only, and `_reject_extra_gate_credential_arguments` rejects stray arguments so that a secret typed on the command line cannot be echoed back in an error.

> **Factory Floor — The directory systemd had reserved.** The boot unit's whole job is to create an empty credential directory under `/run/credentials`. As first written it declared the usual sandbox hardening. The agents' working notes for 2026-09-10 record that it could never see the directory it existed to create: systemd reserves `/run/credentials` and mounts an empty read-only filesystem over it inside any unit that gets its own mount namespace. The cause was found by adding one sandbox property at a time under `systemd-run`. The fix is commit `09360bac3` (authored 2026-09-11), and the unit file on `main` carries the warning in a comment: "Namespace-producing sandbox directives shadow it before ExecStart, so this cold-boot initializer must not use them." Hardening applied by habit can remove the one capability a unit exists for, and the failure looks like an empty directory.

Runners. The operator stages a root-owned source bundle on sefer with a manifest listing every file and its SHA-256. `verify-runner-source-bundle.py` rejects symlinks, unexpected files, unsafe modes and digest mismatches before the first connection to a newborn. Each seat gets its own inference-gateway key with a budget of 200 USD per day (`KEY_MAX_BUDGET_USD`, `KEY_BUDGET_DURATION`), and retirement revokes it.

Controllers. On the hand path inputs cross by the direct pipe described above. On the provider path the runbook says they cross in an encrypted bundle on tmpfs. On both, the tailnet key is passed to `controller-tailnet.py join` on standard input only, and the runbook says it is never restored as a file on a successor. The README adds that the controller's container-gate job keypair is never migrated between controller host generations; each new controller mints its own.

> **Factory Floor — Every reboot re-keyed every host.** On 2026-09-27 the hypervisor exhausted memory and a watchdog reset it (working notes). When the guests came back, the controller and the runners presented new SSH host keys. The cause, found on 2026-10-03, was cloud-init: on these images it deleted and regenerated host keys at boot. Every pinned key in every `known_hosts` file was now wrong, and the only safe way to learn the new ones was through the hypervisor. `sefer-ssh-host-key-refresh.md` records the trust order that came out of it, including the sentence "`ssh-keyscan` is wire evidence, never authority". The fix was a cloud-init drop-in setting `ssh_deletekeys: false`, applied to 17 live guests on 2026-10-03 (README) and installed and verified at every birth since (`e3e549f18`, `install_host_key_persistence`, `assert_host_key_persistence`). An identity proof made at birth is worth only as much as the guarantee that the identity survives a reboot.

12.7. Backups, Blobs and Retention

Disposable machines are safe only because the data is somewhere else. The controller runbook lists what stays external to any controller: the PostgreSQL and Dolt data, the Git remote, the blob store and the credential catalog.

The blob store is content-addressed storage on a NAS mount (`blob_store.py`), used for large artifacts such as logs and evidence. Writes go to a temporary file, are synced and verified, then renamed. If the mount is missing, `put` raises `BlobStoreUnavailable` so that the caller keeps the content inline instead of losing it. Deletion through the library is impossible: `retention_sweep` raises `AppendOnlyPolicyError` with the message "blob retention sweep is permanently disabled by append-only policy". The working notes for 2026-09-22 record the reason. A retention sweep had been deleting blobs that were still referenced. Stephen's ruling was that blobs are never deleted, and a proposal for reference-aware garbage collection was rejected. The code followed the next day (`e7878a248`).

Database backup is packaged in `ops/sefer-backup/` with `postgres_backup.py`: an hourly dump cycle inside the database guest and a separate hypervisor-side check that the off-host VM image is fresh. Defaults are 24 hourly, 7 daily and 4 weekly copies (`DEFAULT_HOURLY_RETENTION`, `DEFAULT_DAILY_RETENTION`, `DEFAULT_WEEKLY_RETENTION`) within 40 GiB (`DEFAULT_BUDGET_BYTES`). The installers never enable or start a unit, and the dump-cycle units carry a condition on an approval marker file. Whether those timers were ever activated in production is not shown on `main`, and the working notes for 2026-10-10 record a possible move toward WAL archiving, so the live backup arrangement is not verified here.

Retention elsewhere is handled by `factory_janitor.py` on a best-effort basis. `should_remove_worktree` marks a task's worktree for removal after 7 days if the task succeeded and 30 if it failed. `yesod export` keeps 30 catalog snapshots (`DEFAULT_KEEP`).

12.8. The Topology Today

The figure and the list below rest on the agents' working notes dated from 2026-10-22 to 2026-11-10 and on runbooks on `main`. Nobody re-observed the deployment for this edition (2026-11-19), so read this section as the best available record of late October and early November, with two specific doubts stated at the end.

- **pompom** is the operator workstation. It runs the CLI and the interactive agents. It is not a service host.
- **sefer** is the large hypervisor, one of three nodes in a cluster, and it runs every factory guest. **seykhl**, which hosted the factory's services in the August edition, no longer hosts factory guests. A third small node is used for pre-runs and caches.
- Guests on sefer: the controller VM; the runner VMs; nine demons, each paired with a test-database container; a container-gate VM for other repositories; the catalog PostgreSQL VM; the Beads Dolt server; the inference gateway VM; a small guest for DHCP and internal DNS; and leased scratch machines.
- **dertog** serves the read-only web and API surfaces.
- **homestar** is the NAS: blob store, harvest archive and VM backups.

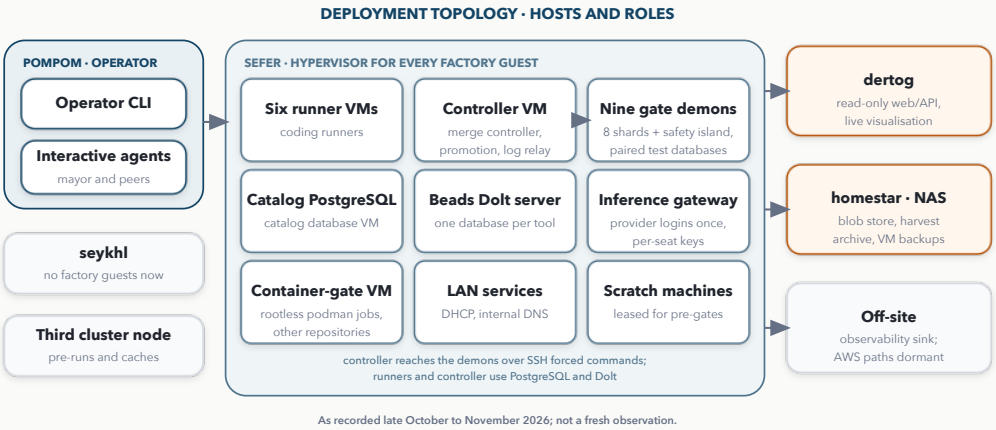

Figure 12.2.: The factory's hosts and guests by role, as recorded in the agents' working notes between late October and November 2026.

- Off site: an observability sink, and AWS for the dormant cloud gate paths.

Two things in that picture are not verified. First, which gate template and recipe generation are live. `main` names VMID 9114 as the default and calls dg6 the current production recipe, the last commit that records the active fleet says dg5 (2026-09-16), and the working notes mention a later template bake for a dg6 set that was born and then blocked. Second, the runner roster. `main` lists six generation 4 seats in `routine-deployment-g4.json`, while the working notes for 2026-11-03 say those runners were stopped and a single runner named gilgul was born. The figure shows runners as a group for that reason.

Birth is the part of the replacement design with the most production history: demons, runners and hand-born controllers have all come from the recipe. The step that has not completed in production is a controller successor enrolling itself (Chapter 11).

Source Notes

Claim or section	Tag	Anchor
Why rebirth: pets, gateway, owner quotes	[notes]	agents' working notes, 2026-08-30, 2026-08-31, 2026-09-22, 2026-09-24
Recipe on one machine	[code]	commit `5f8582b8a`; `ops/yesod-birth/README.md`
Seven steps; newborn has no authority	[code]	`birth-common.sh`; `birth-runner.sh`; `AUTHORITY_UNITS` in `birth-controller.py`

Claim or section	Tag	Anchor
Templates	[code]	README; `DEFAULT_GATE_TEMPLATE_VMID`; `build-gate-template-dg5.sh`; `build-controller-template.sh`; `controller-birth.md`
Generation as recipe label; names	[code]	README "Generation is a recipe label"; `pick-name.sh`; commit `6b488114f`; `Birth.validate`
Controller host generations g1, g4, g5	[deploy], [notes]	`controller-birth.md` (g1, 2026-09-16); working notes 2026-10-26, 2026-11-03
Gate birth trace	[code]	`birth-gate.sh`; `build_dg6_birth_contract`; `gate-profiles.json`; `gate-birth-record.schema.json`; `prepare-gate-readiness.py`; commit `de24c10a9`
Paired database; wrong pairing incident	[code], [notes]	`ops/gate/testdb-birth.sh`; `_paired_database_capability`; working notes 2026-09-18
Inventory prepare, bind, activate	[code]	`generate-gate-fleet-inventory.py`; `manage-gate-fleet-set.py`; README; `sefer-dg6-fleet-sets.md`; `sefer-gate-fleet.json`
Hand-written inventory incident	[notes]	working notes 2026-09-11; commit `339dac3c0`
Runner birth and retirement	[code]	`birth-runner.sh`; `credential-runner.sh`; `retire-runner.sh`; `harvest.sh`
Controller birth, receipts	[code]	`docs/runbooks/controller-birth.md`; `controller-receipts.py`; `ACTIVATION_FILE`
Proven Where	[deploy], [notes]	commit `2584d2844`; `routine-deployment-g4.json`; `controller-birth.md`; `replacement-birth-baseline.md`; working notes
Gate credentials	[code]	`sefer_gate_credentials.py`; `refinery_sefer_credentials.py`; `sefer-gate-credential-lifecycle.md`; `fleet_cli.py`; commit `1ba5312ab`
`/run/credentials` trap	[code], [notes]	unit file `yesod-sefer-gate-credentials.service`; commit `09360bac3`; working notes 2026-09-10
Runner and controller credentials	[code]	`verify-runner-source-bundle.py`; `KEY_MAX_BUDGET_USD`; `controller-birth.md`; README
Cloud-init re-keying	[deploy], [notes]	`sefer-ssh-host-key-refresh.md`; README; commit `e3e549f18`; working notes 2026-09-27

Claim or section	Tag	Anchor
Blob store, append-only rule	[code], [notes]	`blob_store.py` (`AppendOnlyPolicyError`, `BlobStoreUnavailable`); commit `e7878a248`; working notes 2026-09-22
Backup and retention	[code]	`postgres_backup.py`; `ops/sefer-backup/`; `factory_janitor.py`; `backup.py`
Topology	[notes]	working notes 2026-10-22 to 2026-11-10; runbooks on `main`

Not verified:

- Which gate template and recipe generation (dg5 or dg6) serve production today.
- Whether the six generation 4 runner seats on `main` are still the live roster; the working notes for 2026-11-03 say otherwise.
- Whether controller generation numbers are assigned per recipe or per replacement. The README's recipe-label statement names only runners and gate VMs. The relation between controller host generations (g1 to g5) and the three automatic replacement attempts was not reconciled.
- Whether the database backup timers in `ops/sefer-backup/` were ever activated in production.
- The claim that reinstalling the forced command on a born gate breaks the inventory pin rests on the working notes (2026-10-13) and the `regenerate` digest check; the preflight line was not re-read.
- That activation of a gate generation is performed as a drained compare-and-swap; the chapter says only what `sefer-dg6-fleet-sets.md` states.
- The gate birth trace is a reading of `birth-gate.sh` on `main`. No production birth log was read, and how a dg6 set's DHCP reservations are made was not traced.
- The two drift corrections on runners and the 2026-08-31 gateway proof are from working notes only.
- The topology was not re-observed for this edition.

Part IV.

Operating the Factory

13 Seeing the Whole System

This chapter explains how an operator or an agent finds out what the factory is doing, and how far each source of evidence can be trusted. The August edition described one observation framework: an event table in PostgreSQL, an exporter, and a SigNoz backend. That framework was built and its code is still on `main`. It turned out to cover less than it promised, and by November most diagnosis rests on four other streams, one health projection and a handful of read-only commands that did not exist in August or were rebuilt after it.

The chapter also records where the factory was blind this autumn. On 2026-09-09 eight separate failures presented as healthy on the same day. On 2026-09-20 the installed copies of Yesod sat at 0, 35 and 529 commits behind `main` and each one answered questions with confidence. Both episodes changed what the factory measures.

13.1. Telemetry Is Downstream of Truth

One rule from August survived unchanged. PostgreSQL and Git hold the facts. Everything else is a copy that may lag, sample, expire or disappear. The runbook on `main` (`docs/factory-observability-runbook.md`) says it in one sentence: "Never infer a factory transition from the telemetry backend."

> **Invariant — Telemetry is downstream of truth.** A telemetry outage may increase exporter lag. It cannot block a factory transaction or change a result. `factory_events.record_event` does no network work and runs inside the caller's transaction. `FactoryTelemetryExporter.export_once` closes its read connection before it sends anything and opens a new one to advance its cursor. The per-test gate exporters (`gate_test_otel`, `container_gate_otel`) run on daemon threads with a bounded queue (`DEFAULT_QUEUE_CAPACITY = 2048`), and their module docstrings describe the export as a diagnostic copy that leaves the authoritative gate result alone.

The practical consequence is an ordering of trust. A row in `refinery_attempts` outranks an event about that row, and an event outranks a log line. An empty dashboard panel carries no information until someone has shown that the emitter behind it could have fired.

13.2. The Event Spine

The spine is one append-only table, `factory_events`, created by migration `0059_factory_events_ys_yes_rklm.py`. A writer appends an event in the same transaction as the state change the event describes. Each event carries a monotonic `event_id`, a schema version

(`FACTORY_EVENT_SCHEMA_VERSION = 1`), an `event_type` of three lower-case segments, a role, a workflow stage, a source key, the note and root bead it belongs to, an actor, an outcome, a reason code and a payload.

The role says what kind of fact the event is (`EVENT_ROLES`):

Role	Meaning
`intent`	Yesod committed to attempt an effect owned by another system
`transition`	a state change in the same database, committed with the event
`observation`	Yesod observed a fact owned by Git, Beads, a provider or a deployment

There are eight workflow stages (`WORKFLOW_STAGES`: `plan`, `dispatch`, `execute`, `reconcile`, `refinery`, `gate`, `deploy`, `release`) and six outcomes (`OUTCOMES`: `success`, `failure`, `cancelled`, `timeout`, `blocked`, `unknown`). On `main` there are 34 event types. Since 2026-09-10 (`66e7c0a6f`) each type is its own module under `factory_event_taxonomy/`, and the contract is assembled by `discover_factory_event_fragments`, so no central list can drift from the emitters. The families cover planning jobs, dispatch arming and routing, note lifecycle transitions, runs and worktrees, reconciliation of child and root closes, refinery attempts and artifacts, repair lanes, gate executions and cancellations, deployment effects, release observations and backups.

Four properties are enforced by the database or the writer:

- Rows cannot be changed or truncated. Triggers `trg_factory_events_immutable_rows` and `trg_factory_events_immutable_truncate` call `reject_factory_event_mutation`.
- One event per source key. The table has `UNIQUE (source_system, source_event_key)`. `record_event` accepts an identical replay and raises `FactoryEventConflict` on a different event with the same key.
- The payload is allowlisted per event type and capped at `MAX_PAYLOAD_BYTES` (16 KiB). `FORBIDDEN_PAYLOAD_FIELDS` refuses prompts, model responses, tool output, credentials, environment dumps, stdout, stderr and transcripts even if an allowlist were to name them.
- Coverage is declared. `factory_event_coverage` records which emitter promised which stage from which schema version, so a missing event can be read as "before coverage" instead of "did not happen".

Two read models sit on the spine: `GET /api/notes/{note_id}/timeline` and `GET /api/notes/{note_id}/e` backed by `factory_observability.read_timeline` and `explain_note`. Timeline is cursor-paginated (`TIMELINE_DEFAULT_LIMIT = 100`, `TIMELINE_MAX_LIMIT = 500`). Explain returns one of four history statuses (`HISTORY_STATUSES`): `complete`, `partial_legacy` for a chain that predates emitter coverage, `empty_current` when emitters were covered and nothing qualified, and `unknown` when evidence is contradictory or unavailable.

One constraint in the envelope matters for the rest of this chapter: `note_id` and `root_bead_id` are `NOT NULL`. An event must belong to a note.

13.3. The Exporter

One process on the controller, `yesod factory-exporter run`, reads committed events in `event_id` order with `read_event_batch`, turns them into OpenTelemetry logs, spans and metrics, sends them, and only after acknowledgement calls `advance_sink_checkpoint` to move a per-sink cursor in `factory_telemetry_sink_checkpoints`. The cursor never moves backward and never moves without an acknowledgement, so delivery is at least once.

What leaves the database is narrower than what is stored. The log body is `scrubbed_event_envelope`, which copies only a fixed set of envelope fields and filters payload keys again. Spans exist for six event types only (`SPAN_EVENT_TYPES`, named in `BOUNDED_SPAN_NAMES`: `yesod.plan`, `yesod.dispatch`, `yesod.child.run`, `yesod.refinery.attempt`, `yesod.gate`, `yesod.deploy`). Metric labels must come from `METRIC_LABEL_ENUMS`; `validate_metric_labels` rejects anything else, which keeps note ids, SHAs and hostnames out of metric dimensions.

The same process also exports current-state gauges (`export_current_metrics`) and runner diagnostic rows (`export_runner_logs`, with its own cursor).

The first days of export in August produced two emitter bugs of one class. Almost all early events were `reconcile.root_close.observed`, emitted once per reconciler poll over roots that were already closed; that was fixed on 2026-08-17 (`a535f022f`). The first refinery attempt after the emitting release, on 2026-08-16, produced 59 `gate.execution.started` events for one gate execution, one per controller poll (note `ys-yes-bcpn` in `docs/tools/yesod.md` on `main`). An event should record a change or a first observation, and a poll loop deciding to keep waiting is neither.

13.4. The Attempts Nobody Could See

The August edition showed a dashboard panel for refinery and gate events labeled “awaiting first witness” and argued that an honestly labeled empty chart is a coverage statement. A month later a measurement showed that the panel was hiding a structural hole.

> **Factory Floor — 74 attempts, 0 events (2026-09-20).** During the September gate migration, refinery attempts were admitted directly, without a catalog note behind them (the commit calls them “mayor-admitted”, after the coordinating agent). A read-only measurement on the live controller at 2026-09-20T14:36Z, recorded in the docstring of `refinery_failure_journal.py` and in commit `d587d4433`, found 74 attempts created since 2026-09-17. None carried a `source_note_id`. Thirteen reached `gate_red` and six reached `infrastructure_failed`. The same window held 0 `refinery.attempt.*` factory events. The commit names one attempt that had zero records from any service or host over seven days.

The cause is in `refinery_factory_events._attempt_context`. It joins the attempt to its note and returns `None` when the attempt has no `source_note_id` or no root bead. Every attempt and gate projection then emits nothing. This is deliberate: the envelope requires a note, and an attempt without one cannot satisfy it. The effect was that the failures an operator most needed to see during a migration (controller preflight and staging failures on hand-admitted attempts) were absent from the telemetry backend entirely, and nothing said so.

The fix did not widen the envelope. `refinery_failure_journal.journal_attempt_failure` writes one bounded JSON line, event name `refinery_attempt_failed`, to the controller's host journal for each terminal failure edge. `FAILED_ATTEMPT_STATES` lists the edges: `gate_red`, `timed_out`, `needs_rework`, `needs_operator`, `push_raced`, `infrastructure_failed`. The line is written after the durable commit and outside the transaction, from the single fenced choke point in `transition_attempt_state`. Only identity-shaped fields are copied, never free text from a provider error. The line travels on a transport that was already deployed: the machine log collector ships the journal as `service.name=yesod-machine-logs` with the host as a resource attribute, so the failure becomes searchable by attempt and by host.

So "gate events exist in code" and "gate events exist for the attempts in front of you" are different claims. `_project_gate_event` does write `gate.execution.started` and `gate.execution.completed` on the controller, and a red gate carries a bounded `failed_shard_summary` in its evidence (`8afc46ac0`, 2026-09-06). It does so only for attempts that have a note. Whether production attempts carry a `source_note_id` today, and so whether these events now exist in volume, is not recorded in source; the 2026-09-20 measurement is the only one.

13.5. The Four Streams That Carry Diagnosis

Four streams were added in September. None of them is a factory event. All are declared lossy diagnostics, and all reach the backend from the controller or from a collector, never from inside a gate guest.

Stream	Source	Added
Per-test gate logs	`GateTestOtlpExporter`	2026-09-05
Container-gate phases	`ContainerGateOtlpExporter`	2026-09-23
Runner decision logs	`runner_log_telemetry`	2026-09-15
Machine journals and files	host log collector	2026-09-15 on

Per-test gate logs. Each validated test completion in a merge gate becomes one log record of type `pytest.test.completed`, with the gate execution id, the test node id, the outcome and, for failures, a scrubbed failure text limited to 8 KiB and 120 lines (`MAX_FAILURE_BYTES`, `MAX_FAILURE_LINES`). The module docstring of `gate_test_otel` states the boundary: "Gate guests remain credential-free and egress-free." Test records reach the controller through the gate provider's completion path and the controller exports them. The stream answers the blind gate reports of early September, when a red gate with empty per-shard evidence meant the report could not see, and operators read failures off the gate machines by hand (agents' working notes, 2026-09-08 and 09-09).

Container-gate phases (`container_gate.phase` records) do the same for the container gate provider, and **runner decision logs** export the runner's own diagnostic rows as `yesod.runner.log`.

Machine journals and execution files are shipped by a pinned OpenTelemetry Collector (version 0.144.0 per `ops/observability/README.md`) installed as `yesod-log-collector.service` and rendered by `ops/observability/render-log-collector.py`. Gate machines are the narrow exception to "gates have no egress": the collector there runs under a dedicated `yesod-logs` user that is allowed one destination, a relay on the controller, and the README states that gate workers "receive no OTLP configuration, sink credentials, or new egress rights".

Each stream had its own way of failing silently, and the commits record them. Short log files finished between polls and were never read, so readers now start new files at the beginning (`e51a5675a`). With the controller stopped during a direct qualification run, the only caller of the per-test stream poller never ran, and per-test records were invisible for the whole interval; `ShadowCompletionObserver` (`0c5892b71`, 2026-09-20) fills that case and can only stream identities with the prefix `shadow-gate-exec-`, so it cannot take over an admitted gate. The README also warns that a successful probe of the sink's short hostname "does not establish reliable delivery", because the name could resolve through two different search domains (`ce69eed1f`).

13.5.1. One Red Gate, Four Records

A single red gate on an attempt that has a note leaves this trail, in order:

1. The refinery records the gate observation for an attempt whose gate execution row is `red`. In the same transaction `_project_gate_event` appends `gate.execution.completed` with role `transition`, outcome `failure` and a `failed_shard_summary`.
2. While the gate ran, each test completion was queued to `GateTestOtlpExporter` and sent as a `pytest.test.completed` log.
3. The exporter's next `export_once` reads the new event past its cursor, emits a log and a `yesod.gate` span, receives an acknowledgement and advances the checkpoint.
4. The attempt reaches `gate_red`, and `journal_attempt_failure` writes `refinery_attempt_failed` to the controller journal, which the collector ships.

The operator then opens the gate page served by `yesod serve` at `/refinery/gates/<gate_execution_id>`. For an attempt without a note, steps 1 and 3 do not happen, and steps 2 and 4 are all there is.

13.6. One Health Projection

Events answer "what happened to this note". They do not answer "is the factory moving right now". Several operator surfaces need that answer, and each could compute it differently. `factory_stats.collect_factory_stats` (2026-09-10, `b00fac326`) gives them one projection. Its docstring gives the reason: "Keeping the queries and classifications here prevents the operator-facing surfaces from developing subtly different definitions." The CLI (`yesod factory-stats`), the HTTP snapshot, the exporter's gauges, the SigNoz dashboard definition and the Factory Floor page in the live visualization all read the same serialized result (`SCHEMA_VERSION = 2`).

`METRIC_DEFINITIONS` holds 27 gauges. Each one carries a unit, a description and an operator action, so the definition travels with the number. Several of them encode a distinction the autumn paid for:

- A **seat** is a runner, in the sense of one place in the fleet that can hold a claim and run a dispatched agent. `seats_genuinely_working` counts "Live seats with recent branch activity or a live planning lease". A seat that holds a claim and shows neither is not counted as working. The gauge exists because a heartbeat proves that a thread is alive and says nothing about its work loop.
- `idle_seats_while_missing_claimable_work_exists` counts idle seats while ready work is absent from the claimable read model. It names a failure in which every part reports healthy and nothing runs.
- `seconds_since_last_merge_to_main` is computed from the authoritative merge summary, because refinery merge commits carry synthetic dates (agents' working notes).
- `gate_outcomes_last_n` keeps test progress counts nullable. Its action text tells the operator to treat "missing or partial counts as infrastructure evidence", so an unknown is not read as zero failures.

The alert thresholds are a dataclass, `FactoryStatsThresholds`, described in source as "Incident-derived defaults, all overridable at runtime":

Signal	Warning	Critical
Time since last merge	3,600 s	7,200 s
Awaiting-merge depth	6	10
Time since last gate start	1,800 s	5,400 s
Working seats (below)	50%	0%
Critical dependency depth	3	5
Oldest claimable work	7,200 s	21,600 s
Fenced active planning jobs	1	1

Two alarms fire from code and deliver by durable mail (`refinery_alarm_delivery.py`): `refinery_alarm.py` for repair and dispatch starvation (2026-09-11) and `refinery_lane_alarm.py` for refinery lane and planning progress (2026-10-13, `a7fed63db`). The second was merged the day before the date the agents' working notes give (2026-10-14) for a refinery lane deadlock that went about 25 minutes unnoticed; whether the alarm was deployed by then is not recorded. The notes record Stephen's decision after it: alert logic belongs in code that reads PostgreSQL and Git and computes invariants, not in dashboard queries. The lane alarm is the nearest thing to that on `main`; a fuller set of independent checker processes exists in the notes only.

`yesod why-not-running NOTE_ID` answers the single most common question, why an armed note has not been claimed. A first version dates from 2026-07-20; it was rebuilt on 2026-09-10 (`4fba569d0`) on a read-only evidence layer, `why_not_running.py` (`5085b1d73`). The docstring of `diagnostics.py`, where the command lives, lists what it checks: the bead, readiness, snapshot freshness, status, disposition, workspace and lane.

13.7. Deployment Observation

Whether merged code is running is a separate observation problem, and Chapter 10 (Merged Is Not Deployed) owns it. Two instruments matter here. The deployment observer (`deployment_observer.py`, "the three deployment truth surfaces") writes to the append-only `deployment_surface_observations` table and feeds five gauges in the health projection: `deploy_lag`, `seats_current`, `birth_drift`, `config_drift` and `done_not_deployed`.

The second is `yesod release-skew` (2026-09-21, `7bb582e12`). Chapter 10 tells the episode behind it: on 2026-09-20 the build seats, the controller and the operator CLI sat 0, 35 and 529 commits behind `main` at the same time, each answered confidently, and no command reported the distance. For observation the lesson is in two types. `SurfaceObservation` and `Distance` cannot be constructed unidentified without a reason, so an unreachable seat is reported `unreachable` with a reason such as `probe_timeout` and is never counted as current.

13.8. Eight Silent Failures in One Day

The September episode is recorded only in the agents' working notes for 2026-09-09, and the count is theirs. On that day eight conditions were found, each presenting as healthy:

- a refinery lane slot held by an attempt that was already terminal, so nothing was claimable for an hour while the controller reported `tick_error_count: 0`;
- cloud-init rewriting the hosts file on the controller, so eight of nine gate hosts did not resolve;
- gate database credentials stored on tmpfs and destroyed by a reboot;
- PostgreSQL down on eight of nine gate databases while the distribution's umbrella service unit reported `active`;
- latency in the claimable read model leaving approved work unclaimed;
- `runs.result_commit` empty for every run since late August, which made "produced no commits" meaningless as evidence;
- a run's `max_runtime_seconds` recorded and never enforced;
- dispatch runs unmetered for cost since 2026-08-10.

The notes draw one lesson: the absence of a bad signal was being read as a good one. The measurement work that followed answers that directly: the health projection, the genuinely-working seat gauge, the rebuilt `why-not-running` and the Factory Floor page were all merged to `main` on 2026-09-10. The gate credential problem was fixed in `a69954af0` and belongs to Chapter 12 (Born, Not Repaired). The runtime item needs a qualification from source. On `main` the owning runner's wait loop, `_wait_with_governance`, enforces the wall-clock limit on the agent process it launched and returns `timeout`, and that loop was already in `runner.py` before 2026-09-09. What has no automatic caller is a kill request from outside the runner: `run_ledger.request_kill` is called only by the CLI verb `yesod runs kill`. The notes' "never enforced" is what they recorded on that date.

The same shape recurred in the agents' own tooling in October, outside product code. The working notes for 2026-10-11 to 10-16 describe a watcher cursor advanced before delivery (the opposite of `advance_sink_checkpoint`) and a sentinel that printed `HELD` for about four

hours while every hold call it made was failing. On 2026-11-08 a planning queue that looked frozen was a stopped planner VM, down for 15 hours 45 minutes; job-state snapshots said nothing about the machine.

13.9. The Backend

> **Proven Where — SigNoz, as of 2026-11-19.** *In code on `main`:* the exporter, the four side streams, the collector renderer and installer, a `yesod-factory-health` dashboard definition (`ops/observability/factory-stats-dashboard.json`) and its provisioner. *Run in production:* the exporter was commissioned on 2026-08-16 and delivered the event history to a temporary SigNoz host; a 24-hour census on that host is cited in commit `140a3bc8d` (2026-09-20), so it was receiving then. *Not established:* the runbook on `main` (last changed 2026-10-07) still calls that host "the temporary target" and still names the retired controller host. No source records a move to the planned permanent host. Applying the factory-health dashboard is "a separate, deliberate human operation" in the runbook, and no source records that it was done. Whether the two August dashboards and the exporter-lag alert still exist is not recorded, which is why this edition does not reprint their screenshots. Whether collectors are installed on every host born in November is not recorded either; the README warns that "a live installation does not update a template". *Dormant:* a second, hosted profile for an operator-run SigNoz on EC2, whose verification record starts at `status: not_run`.

13.10. Compliance as a Query

On 2026-10-23 the organization's operating rules became a read-only registry, `yesod rules check` and `yesod rules show`. Twelve rules, K1 to K12, each have an owner and a stated assurance level; six have a detector (assurance `detected`) and six are attested only; those six make up `NEVER_CHECKED_RULES`, each with a recorded reason it cannot yet be checked. Chapter 17 (The Agent Organization) describes the rules and what each detector can and cannot see.

13.11. The Evidence Map

The operator still picks the surface that owns the question.

Question	Primary evidence
What was requested, what was planned?	note, comments, plan revision
Why is an armed note not running?	`yesod why-not-running`
What did one run do?	`runs` row, session archive
Did it produce a branch?	Git remote ref, result commit
Where is integration?	`refinery_attempts`, refinery lane state

Question	Primary evidence
What failed in the gate?	gate page, per-test logs
Did the commits land?	`origin/main` ancestry
Is merged code running?	`yesod release-skew`
Is the factory moving?	`yesod factory-stats`
What happened to this note, in order?	timeline and explain

The run ledger changed in one respect that matters here. A `runs` row now exists from the claim, in status `starting` (2026-10-13, `98f3b8262`), and rows that a dead runner left live become `lost` in `reconcile_lost_runs`. Chapter 6 (The Runner) covers the ledger.

> **Terminology Trap — Attempt state is not note status.** A note can be `awaiting_merge` while its refinery attempt is `gate_red`, or `blocked` while its attempt is `merged_not_deployed`. Reading only one side produces false conclusions.

13.12. A Read-Only Diagnostic Rhythm

The order from the August edition still holds: diagnose a stuck item in pipeline order, and stop at the first surface that disagrees with the one before it.

1. Identify the note and its root bead; read the note status and disposition.
2. Run `yesod why-not-running` for the note if it is armed and unclaimed.
3. Check that the machine that should be acting is running at all.
4. Inspect live runs and the last terminal run for the note.
5. Inspect the remote branch and the result commit.
6. Inspect the refinery attempt and whether admissions are paused.
7. Inspect the gate page and the per-test failures.
8. Check `origin/main` ancestry.
9. Check release skew before saying anything is deployed.

Step 3 was added after 2026-11-08.

```
yesod factory-stats
yesod why-not-running NOTE_ID --json
yesod codefactory-dispatch status
yesod runs list
yesod runs failures
yesod refinery status
yesod refinery admissions status
yesod release-skew --strict
yesod rules check
```

These verbs read state. They still open a connection to the production database, and Chapter 10 explains why an operator CLI that is behind the schema must never be allowed to migrate it.

Operator Check — Preserve the scene. Before a reset, kill, dequeue or retry, capture the run row, the attempt row, the branch state, the gate page and the relevant service status. Then check whether the item is in the state where the machine is about to act on it. Chapter 19 (What Failed, August to November) tells of four repairs in September that looked stranded and were in fact pulled out of a claimable state by the recovery itself.

Source Notes

Claim or section	Tag	Anchor
Telemetry downstream of truth	[code]	`factory_events.record_event`; `FactoryTelemetryExporter.export_once`; `gate_test_otel.py`, `container_gate_otel.py` docstrings; `docs/factory-observability-runbook.md`
Event spine, envelope, invariants	[code]	migration `0059_factory_events_ys_yes_rklm.py`; `factory_events.py` (`record_event`); `factory_event_taxonomy/` (34 modules); `66e7c0a6f`
Timeline and explain	[code]	`factory_observability.py` (`read_timeline`, `explain_note`, `HISTORY_STATUSES`); `live_viz_api.py`
Exporter and projection limits	[code]	`factory_telemetry.py` (`advance_sink_checkpoint`, `scrubbed_event_envelope`); `factory_telemetry_cli.py`
August emitter bugs	[code], [deploy]	`a535f022f` (2026-08-17); note `ys-yes-bcpn` in `docs/tools/yesod.md` (2026-08-16)
74 attempts, 0 events	[code], [deploy] 2026-09-20	`refinery_failure_journal.py` docstring; `d587d4433`; `refinery_factory_events._attempt_context`
Four side streams and their traps	[code]	`gate_test_otel.py`, `container_gate_otel.py`, `runner_log_telemetry.py`, `shadow_test_stream.py`; `ops/observability/README.md`; `e51a5675a`, `0c5892b71`, `ce69eed1f`, `b81ba9fee`
Blind gate reports	[notes]	agents' working notes, 2026-09-08/09
Health projection, seat, thresholds	[code]	`factory_stats.py` (`collect_factory_stats`, `METRIC_DEFINITIONS`, `FactoryStatsThresholds`); `b00fac326`; Factory Floor page `c9c6d8192`

Claim or section	Tag	Anchor
Alarms; `why-not-running`	[code]	`refinery_alarm.py`, `refinery_lane_alarm.py`, `refinery_alarm_delivery.py`, `a7fed63db`; `diagnostics.py`, `why_not_running.py`, `4fba569d0`, `5085b1d73`
Lane deadlock unnoticed; checker decision	[notes]	agents' working notes, 2026-10-14
Deployment observation; release skew	[code]	`deployment_observer.py`; `release_skew.py` (`SurfaceObservation`, `Distance`); `7bb582e12`, `308628d7d`
Eight silent failures	[notes]	agents' working notes, 2026-09-09; `a69954af0`
Runtime limit enforcement	[code]	`runner.py` (`_wait_with_governance`, `_effective_max_runtime`); `run_ledger.request_kill`, called only from `runs_cli.py`
Watchers, sentinel, stopped VM	[notes]	agents' working notes, 2026-10-11 to 10-16 and 2026-11-08
SigNoz status	[deploy]	`docs/factory-observability-runbook.md` (last changed 2026-10-07, `111d8f5cd`); `140a3bc8d`; `ops/observability/`
Compliance registry	[code]	`rules_contract.py` (`RULES`, `NEVER_CHECKED_RULES`), `rules_cli.py`; `30cc51daf`
Run ledger	[code]	`run_ledger.py` (`reconcile_lost_runs`); `runs_cli.py`; `98f3b8262`
Diagnostic commands	[code]	`factory_stats_cli.py`, `diagnostics.py`, `dispatch_cli.py`, `runs_cli.py`, `refinery_cli.py`, `cli.py` (`release-skew`), `rules_cli.py`

Not verified:

- Whether production refinery attempts carry a `source_note_id` today, and so whether `refinery.*` and `gate.*` factory events now exist in volume. Only the 2026-09-20 measurement is in source.
- Whether the gate emitter noise recorded in `ys-yes-bcpn` was fixed; no commit on `main` cites that note id.
- The current SigNoz host, whether the `yesod-factory-health` dashboard was ever applied, and whether the two August dashboards and the exporter-lag alert still exist. The three August screenshots were dropped for this reason.
- Whether log collectors are installed on every host born in November.
- The count of eight failures on 2026-09-09, the 25 minutes on 2026-10-14, the four hours of the false `HELD` sentinel and the 15 hours 45 minutes on 2026-11-08 come from the agents' working notes only.

- What the 2026-09-09 notes meant by a runtime limit "never enforced". The runner's own wait loop enforced it in source before that date; the notes may refer to runs whose runner was gone, and that reading is inferred.
- Whether the lane alarm was deployed on the controller by 2026-10-14.
- The add dates of the four streams are the commit dates of their first modules on `main`, not deployment dates.
- The definition of "seat" is assembled from the metric descriptions in `METRIC_DEFINITIONS`; source has no standalone definition.

14 Failure Is a State Machine

This chapter is a reference. It sorts the ways work fails in Yesod into seven classes and gives, for each one, the exact states that mean "failed", the reason codes worth knowing, and the supported way out. Each class has one short dated example. Chapter 19 (What Failed, August to November) tells the stories in order; this chapter is the map to read them with.

The design idea is unchanged from the first edition. In a factory run by agents nobody is watching, so a stuck piece of work has to be a row that someone can query, with a name, an entry condition and a defined exit. What the autumn of 2026 added is a ranking. The classes with rich state machines failed often and cheaply. The classes with weak states, or none, cost most of the lost time.

14.1. What Held and What Did Not

The August edition summarized the factory in one sentence: the components fail constantly and the guarantees must not fail once. After three more months that sentence has to be split in two.

Integrity held. The research for this edition found no record, between 2026-08-20 and 2026-11-19, of a wrong candidate reaching `main` through the gate, and no record of corrupted lifecycle rows. This is inferred from the incident chronicle and the commit history; it is an absence of evidence, not a proof. The breaks to `main` that are on record went around the gate. On 2026-09-11 an operator merged one note without a green gate attempt and said so in the merge commit (`8326d9a3c`; Chapter 9, The Deterministic Gate), and the repair commit of 2026-09-14 (`0373aa5b4`; Chapter 1, Follow the Change) records that pull requests merged through GitHub without the gate had left `main` unable to pass its own gate.

Availability did not hold. The factory was stopped repeatedly, sometimes for many hours, by failures that no state named. The agents' working notes record a refinery lane that did nothing for four hours on 2026-09-09 while the controller reported zero tick errors, about nineteen hours with every runner wedged on 2026-10-07 and 08, and a planner VM stopped for 15 hours 45 minutes on 2026-11-08 while two agents reasoned about job states. None of these corrupted anything. All of them stopped the work.

Four kinds of failure account for most of that lost time, and the vocabulary of August has no word for any of them:

- A component that is alive and doing nothing. Its evidence is not wrong. Evidence of progress is absent, and nothing is required to notice.
- Skew between surfaces. Code, schema, configuration and installed release move at different times.

- Replay hazards. A heal or retry re-executes something that was correct when first run and is wrong now.
- Coordination failures between agents and the one human operator.

The seven classes below are ordered roughly from the most mechanism to the least.

14.2. The Vocabulary

A few terms are used the same way in every class.

A note carries two independent fields. Its **lifecycle status** (`tool_notes.status`, twelve states in `IMPLEMENTATION_LIFECYCLE`) says where the work is. Its **disposition** (`disposition.py`: `active`, `held`, `blocked`, `needs_rework`, `quarantined`, `deferred`) says whether automation may act on it. Every disposition except `active` vetoes claim, gate enrollment and merge (`disposition_eligible_sql`).

`awaiting_verification` is the lifecycle status for an execution that ended without provable success. The note parks there with a structured reason and nothing else advances.

On the merge side a failure is a **terminal attempt**. `allowed_attempt_transitions` returns an empty set for every terminal `AttemptState`, so a retry is always a new row linked by `prior_attempt_id`.

When recovery would require a guess, the controller opens an item in the **operator inbox** (`yesod refinery inbox`) and waits for a decision.

> **Terminology Trap — `blocked`.** The August edition called `blocked` a lifecycle state. It is a disposition. The twelve-state lifecycle contract has no `blocked`. The confusion has a real source: `Runner._handle_failure` and the refinery's `_project_merged_not_deployed` still write the raw string `blocked` into the status column alongside the `BLOCKED` disposition, and `note_status.is_terminal` describes it as a legacy value that stays readable. Read the disposition, and read its reason.

> **Terminology Trap — `merge_conflict`.** The August edition listed `merge_conflict` as a terminal attempt state. It is a reason code. A conflicting candidate ends the attempt in `needs_rework` with `reason_code=merge_conflict`. The terminal set is `TERMINAL_ATTEMPT_STATES`, ten states, listed under class 3.

14.3. Class 1: Execution

This is one agent run on one note, and it has the richest failure states because an agent can fail in the most ways.

The runner records two things about a failed run. The **verdict** (`reconciliation.py`) says what happened to the work: `work-complete-unclaimed`, `work-present-uncommitted`, `partial-progress`, `no-progress`, `infra` or `needs-reconcile`. The run's terminal status in the run ledger (`run_ledger.TERMINAL_STATUSES`) says how the process ended: `success`, `failure`, `killed`, `timeout`, `lost`, `token_budget`. The two are kept apart because the right next step

depends on the first, and a blind retry of `work-present-uncommitted` would destroy the evidence.

Failed states: note status `awaiting_verification` with a reason; disposition `needs_rework` or `blocked`; run status `lost`. For planned work the reason code is `root_dispatch_child_reconciliation_fai` with one of the `FAILURE_CODES`, such as `result_commit_missing` or `child_close_failed`. A child that ends with no commit must emit one `YESOD_CHILD_OUTCOME:` record (`blocked_on_dependency` or `no_changes_needed`); silence is `child_outcome_missing`, which reopens the child and sets `needs_rework`. A third outcome, `acceptance_not_executed`, belongs to a child that committed and whose prescribed tests did not run (Chapter 15, The Reliability Contracts).

Retries are bounded. `DEFAULT_MAX_RETRIES` is 3 with backoff that doubles per attempt. Two consecutive `no-progress` verdicts stop early: `_handle_failure` records `retry_halted_no_progress`, where a spent budget records `retry_exhausted`. Its docstring says that `infra` verdicts never count toward that guard, because an environment failure says nothing about the task.

The ways out all re-prove the world before reopening anything. `planned-root-retry` handles a first child that produced no commit. `retained-child-reconcile` handles a first child that committed and could not push. `completed-plan-reconcile` handles a note whose children are all closed and whose lifecycle never advanced. `reconcile_lost_runs` and the exact-owner repair of `84eee205f` handle a run whose owner died.

Example. On 2026-09-16 a first child committed its work and the push failed. `planned-root-retry` refuses that case, because retrying would discard a real commit. The response was a new verb, `retained-child-reconcile` (`c3be1837f`, runbook `retained-child-recovery.md`), which journals an intent, pushes the retained commit under a lease and resumes the remaining children.

14.4. Class 2: Planning

A plan is produced by a durable job in `planning_jobs`, and the job has its own states: active `queued`, `running`, `retrying`, `blocked`; terminal `succeeded`, `failed`, `cancelled`, `superseded`.

The terminal reasons seen in source are `stale_note_revision` (the note's planning input changed under a running job), `note_revision_advanced`, `completed_prefix_invalid`, `invalid_provider_output` and `planning_root_bead_missing`. The agents' working notes add that a plan returned with open questions fails the same way; that path was not traced in source. A job that uses up its attempts ends `failed`, and the note is held with a reason that starts `planning_retry_exhausted:` or `planning_invalid_output:`. A job whose note is not `active` is never leased; the classifier reports `planning_disposition_blocks_lease`.

The trap in this class is that the note's status is a projection. A note can show `planning` while every job it has is terminal. The supported inspection path is `note-plan-status` and the runbook `planning-job-inspection.md`. The `note-plan` verb is a mutating request, and running it to look at a plan supersedes the plan.

Ways out: `planning-retry` for a failed job; `replan` for a note whose disposition is `needs_rework` or `blocked` (it refuses `held`, `quarantined` and `deferred` with `disposition_not_recoverable`); for open questions, write the answers into the note body and plan again.

Example. According to the agents' working notes for 2026-09-11, one note lost two planning job generations to comments written to help it. Comments are part of the planning input fingerprint, so each comment ended the running job with `stale_note_revision`, and the note went on showing `planning`.

14.5. Class 3: Integration

This is the merge, owned by the refinery controller (Chapter 8, The Merge Controller). Failure here is a sealed record. The ten terminal states are `succeeded`, `gate_red`, `superseded`, `cancelled`, `timed_out`, `needs_rework`, `needs_operator`, `push_raced`, `infrastructure_failed` and `merged_not_deployed`.

Terminal state	Meaning	Way out
`gate_red`	the gate ran and said no	fix the branch, then `rearm`
`push_raced`	the target moved before the push	a new attempt on the new base
`needs_rework`	conflict, or a failed pre-gate	conflict repair, or rework
`needs_operator`	a rule could not decide	resolve the inbox item

A conflict creates one row in `refinery_conflict_repairs`. Its active states are `requested`, `armed`, `running` and `awaiting_gate`. Arming returns the note to `open`, a repair agent merges the target into the branch, and the repaired branch enters as a fresh attempt through the full gate. A held or terminal source note cancels an unsubmitted attempt with `note_no_longer_eligible`.

Two corrections to the August text belong here. First, refinery lanes isolate repositories from one another, and inside one the order is strict first-in, first-out by attempt number, so one slow attempt at the head delays everything behind it. Priority does not reorder a refinery lane. Second, a red gate does not retry itself. The supported re-entry after fixing the branch is `yesod refinery rearm`.

The alive-but-idle failure showed up here first, and two mechanisms now name it: `reconcile_terminal_lane_slots` clears a refinery lane slot still pointing at a finished attempt, and `_requested_without_execution_stall_error` turns "requested attempts, no live gate, not paused" into a tick error after `STALL_GRACE_MINUTES` (15).

Example. The agents' working notes for 2026-09-09 record three notes sitting in `awaiting_merge` behind terminal `gate_red` attempts, one for about 23 hours. Nothing was broken. Each was waiting for a re-entry that no component was going to perform.

14.6. Class 4: Gate

"Gate" here means the merge-time test gate (Chapter 9). A gate execution has its own states in `GateExecutionState`: `submitted`, `starting`, `running`, then `green`, `red`, `timed_out`, `cancelled`

or `lost`.

The line that matters is between a verdict and an infrastructure failure. `red` is a verdict about the candidate and ends the attempt `gate_red`. `timed_out`, `cancelled` and `lost` say nothing about the candidate; the attempt ends `timed_out`, `cancelled` or `infrastructure_failed`. On the home gate fleet a worker that fails its preflight reports `worker_rejected` (`refinery_sefer_gate.py`), which is also an infrastructure answer. There is no silent fallback to another provider.

The way out of a verdict is a changed branch and a new attempt. The way out of an infrastructure failure is to repair the fleet, by the runbooks in class 6, and submit a new attempt for the same branch.

Two newer mechanisms sit in front of the gate. The pre-gate (`refinery_pre_gate.py`, modes `off`, `shadow`, `active`) is on `main` and is not enforcing: the reviewed controller document has no pre-gate key, so the code default `off` applies. The "pre-run" is a practice, not code: since 2026-10-13 the agents hold every delivery until a full test pass of its exact tip is clean on a disposable box (agents' working notes).

Example. A commit of 2026-09-20 (`cfb66cf03`) records that three readers of the pytest report log ignored failed subtests, so each could count a red run green. The fix made a failed subtest turn its parent red in every reader.

14.7. Class 5: Deployment and Replacement

After the push, an attempt on a refinery lane with a deployment step is `deploying`. If the independent observer reports the exact candidate, it ends `succeeded`. If the observer stays pending past the settle budget, or fails, it ends `merged_not_deployed` and the controller opens an operator item. The note is projected to `blocked`, and notes that depend on it do not unblock on their own. Chapter 10 (Merged Is Not Deployed) owns this state, its reason codes and its retry contract.

Skew has named states too (also Chapter 10). `yesod release-skew` reports each install as `current`, `behind`, `ahead`, `diverged`, `unknown_release`, `unknown_distance` or `unreachable`.

A controller replacement (Chapter 11, Deploy Is a Host Swap) is a second state machine. A replacement intent ends `succeeded`, `rolled_back` or `needs_operator`; a target host can end `failed` or `stopped_retained`. Automatic rollback is legal only in a few phases, and `ControllerReplacementHandover.rollback` refuses once the release of the service addresses has been claimed. After that point the way out is an operator following `replacement-driver.md`.

> **Proven Where — as of 2026-11-19.** The replacement states are on `main`. No replacement intent has reached `succeeded` in production. The three production attempts at a driver-born successor ended needing an operator before handover, so `rolled_back` has never been produced in production either. The way out of `merged_not_deployed` has so far always been an operator running a deployment by hand.

Example. The fix note for the successor's first stage passed the gate and its attempt ended `merged_not_deployed`, because the serving controller still ran the older release (agents' work-

ing notes). The merge is the candidate commit `16606a42` on `main` (stamped 2026-11-17 19:16Z, a time derived from its source commit, not the time of landing); the working notes place the merge window on 2026-11-18. The fix for the swap is on `main` and running on no controller.

14.8. Class 6: Host and Infrastructure

There is no single state machine for a host. The evidence appears in whichever class the host serves: a runner row marked `dead`, a gate preflight answering `worker_rejected`, an attempt ending `infrastructure_failed`. Often it appears as nothing at all, which is the alive-but-idle failure again.

The causes on record in the agents' working notes are ordinary: host memory exhaustion, SSH host keys regenerated by cloud-init on reboot, gate credentials kept on tmpfs and lost at reboot, expired gateway keys, an expired tailnet node key.

Recovery is by runbook, and the runbooks are on `main` under `docs/runbooks/`: `sefer-ssh-host-key-refresh.md`, `sefer-gate-credential-lifecycle.md`, `sefer-gate-headroom-advisory.md`, `sefer-dentry-relief.md`, `vm102-postgres-backup-automation.md`. The longer answer is to stop repairing hosts and replace them (Chapter 12, Born, Not Repaired).

Example. The agents' working notes record that on 2026-09-27 at 17:52Z the hypervisor sefer ran out of memory and was reset by a watchdog. Runners and gate VMs stayed stopped after the reboot, and every VM came back with new SSH host keys. No row anywhere said "host down".

> **Operator Check — is the worker alive?** Before reasoning about why a queue is not moving, list the runner rows, including the `dead` ones, and confirm the VM is running. On 2026-11-08 this check would have saved most of a day.

14.9. Class 7: Coordination

This class has no states in the database, and that is the finding. The durable artifacts are few: a `held` disposition with a reason, the admissions pause with its actor and revision, a signed grant from Stephen (Chapter 18, Signed Human Authority), and receipts.

The failure modes are a wrong cause posted as fact, a recovery that moves a note out of the state where the machine could have finished it, an approval exercised while a release window is open, and a process step that stops when the agent session that owned it ends. Recovery is procedural: trace before naming a cause, check whether the machine is about to act before recovering, and give every recurring step an owner that outlives a session.

This class has the weakest mechanism and, by the record in the working notes, caused a large share of the lost time.

Example. Between 2026-09-09 and 09-11 four conflict repairs looked stranded. According to the agents' working notes the status history showed that arming had worked each time, and a recovery procedure had then pulled each note back into planning, where an armed repair can never be claimed. Two were frozen for about 40 hours by the recovery.

14.10. Reading Across the Classes

Class	Where failure is recorded	Way out
1 Execution	note status, disposition, run ledger	re-proving recovery verbs
2 Planning	`planning_jobs` state and reason	`planning-retry`, `replan`
3 Integration	terminal attempt	a new attempt
4 Gate	gate execution, attempt	new branch, or repaired fleet
5 Deployment	attempt, skew, intent	next deployment; operator
6 Host	other classes, or nowhere	runbook; replacement
7 Coordination	holds, pause, grants	procedure

Read down the middle column and the amount of mechanism falls. The first four classes absorb failure into named states, and those are the classes where integrity held. The last three are where the factory stopped. The changes that came out of the autumn mostly give those three something to record: `merged_not_deployed`, the skew statuses, the stall sentinel, the owned admissions pause. Chapter 15 states the rules behind those changes.

Source Notes

Claim or section	Tag	Anchor
Integrity held, availability did not	[inferred]	incident chronicle in research brief 10; commits `8326d9a3c`, `0373aa5b4`; agents' working notes, 2026-09-09, 10-08, 11-08
Lifecycle and disposition	[code]	`note_lifecycle.py` `IMPLEMENTATION_LIFECYCLE`; `disposition.py`; `disposition_eligible_sql`
`blocked` written to status	[code]	`runner.py` `_handle_failure`, `_write_failure_note_state`; `refinery_intake.py` `_project_merged_not_deployed`; `note_status.py` `is_terminal`
`merge_conflict` is a reason	[code]	`refinery_state.py` `TERMINAL_ATTEMPT_STATES`; `refinery_conflict_repair.py`

Claim or section	Tag	Anchor
Class 1	[code]	`reconciliation.py` `VERDICT_*`; `run_ledger.py`; `root_dispatch_reconciliation.py` `REASON_CODE`, `FAILURE_CODES`; `child_outcomes.py`; `dispatch_cli.py`; `84eee205f`, `c3be1837f`
Class 2	[code], [notes]	`planning_jobs.py`; `planning_publication.py`; `planning_execution.py`; `note_lifecycle_ops.py`; `docs/runbooks/planning-job-inspection.md`; agents' working notes, 2026-09-11
Class 3	[code], [notes]	`refinery_state.py`; `refinery_control.py`; `refinery_conflict_repair.py` `ACTIVE_REPAIR_STATES`; `refinery_runtime.py`; `refinery_queue.py` `STALL_GRACE_MINUTES`; agents' working notes, 2026-09-09
Class 4	[code], [deploy]	`GateExecutionState`; `refinery_sefer_gate.py`; `refinery_pre_gate.py`; `controller-reviewed-document.json` (no pre-gate key); `cfb66cf03`; agents' working notes, 2026-10-13
Class 5	[code], [notes]	`refinery_controller.py`; `release_skew.py`; `replacement_contract.py`; `replacement_handover.py`; `docs/runbooks/replacement-driver.md`; `16606a42`; agents' working notes, 2026-11-18
Class 6	[code], [notes]	runbook files under `docs/runbooks/`; `runner.py` (runner status `dead`); agents' working notes, 2026-09-27, 11-08
Class 7	[notes]	agents' working notes, 2026-09-11, 10-28, 11-09; `7c8ff9b41`

Not verified:

- The integrity statement rests on the absence of a recorded bad merge or corrupted row in commit history and the working notes. Nobody audited the production database for this edition.
- Whether any candidate landed on the strength of a red run counted green before `cfb66cf03` was not traced.
- State lists come from enum and constant definitions on `main`. Transitions were not traced end to end through `runner.py` and `refinery_controller.py`.
- The terminal states of `refinery_conflict_repairs` and the full set of planning terminal reason codes were not enumerated from source; only the names printed here were confirmed.

- That a planner output with open questions ends as `invalid_provider_output` comes from the agents' working notes; it was not found in the planning modules read.
- Whether an `infrastructure_failed` attempt is ever retried without an operator or a re-enqueue was not traced.
- Durations and counts attributed to the agents' working notes (four hours, nineteen hours, 15 hours 45 minutes, 23 hours, 40 hours) were recorded by agents at the time and have no corroborating measurement on `main`.

15 The Reliability Contracts

This chapter names the pattern that the machinery in Chapters 5 to 12 keeps repeating. It starts with the smallest example, a planned child that exits cleanly having produced nothing, and draws five rules from it. Those five rules are the August 2026 edition's, and they still describe the code. The chapter then keeps the August promotion campaign as dated history, adds a footnote that November forced onto the first contract, and adds five more rules that the autumn paid for, each with the symbol that enforces it or a plain statement that nothing does.

A **reliability contract** is a boundary where the factory states what evidence must exist before state may advance, and what happens when that evidence is missing. Chapter 14 (Failure Is a State Machine) lists the states work falls into. This chapter is about the rules that decide when it falls.

15.1. The Simple Example That Started It

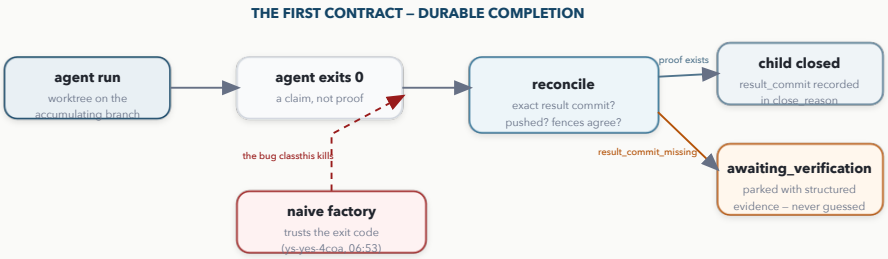

Figure 15.1.: A planned child is complete only when its exact result commit is on the pushed branch and the child bead is closed with a reason that names that commit.

Consider the smallest version of the problem: one planned task, one agent, one branch. The runner launches the agent in a worktree. The agent works and exits with status zero. A naive factory treats exit zero as success and closes the task.

One morning in the summer of 2026 a coding agent ran for ten minutes on the second child of a planned note, logged nothing, produced zero commits, and exited cleanly. A naive factory would have closed that child, launched the third on top of nothing, and merged a feature that half existed. Nothing downstream would have looked back.

The first reliability contract is the refusal to let that happen:

> **Invariant — Durable completion.** A child is complete only when an exact result commit exists on the accumulating branch, is pushed, and every identity check agrees. `validate_leaf_result_commit` requires a captured base SHA, a result commit equal to the worktree's exact `HEAD`, the base as an ancestor of the result, and at least one commit between them. `reconcile_successful_child` then closes the child bead with a canonical reason and reads it back. An agent's exit status is a claim. The commit is the proof.

When the proof is missing, the run is reconciled as a failure. The note parks in `awaiting_verification` with reason code `root_dispatch_child_reconciliation_failed`, a failure code such as `result_commit_missing`, the run id, the child id and the plan fingerprint. No third child starts and nothing is enqueued for merge.

Every later contract repeats this move. It takes an assumption ("an agent that exits zero succeeded"), replaces it with an evidence requirement (a result commit), and makes the missing-evidence path a parked, inspectable state. The factory's word for the discipline is **fail closed**.

15.2. The Five Rules, Named

Reduce each contract in the book to its moves and the same five appear.

1. **Name the evidence.** State advances only when a specific artifact exists: a result commit, a marker hash, a process identity, an applied-migration set. A signal that suggests success does not count.
2. **Fail closed with structured evidence.** When the evidence is missing, ambiguous or contradictory, refuse, and make the refusal say what disagreed with what.
3. **Journal intent before effect.** Anything that changes the world writes a durable intent row before the change and an observed row after it, so a crash between the two is recoverable and a replay is idempotent.
4. **Re-prove instead of remembering.** Recovery recomputes the world from durable state (Git remotes, process tables, database rows) and never from a cached phase.
5. **Bind to exact identity.** Work is bound to exact SHAs, plan fingerprints, note revisions, idempotency keys and fencing generations. A near match must not match.

All five have already appeared. The merge controller's fencing tokens (`LeaseAuthority`; Chapter 8, The Merge Controller) are rule 5, and its lease-based reclaim is rule 4. The gate's line between a red verdict and an infrastructure failure (Chapter 9, The Deterministic Gate) is rule 2. The replacement effect journal (`replacement_effect_journal`; Chapter 11, Deploy Is a Host Swap) is rule 3. Every park on a missing commit is rule 1. The newer subsystems built since August (controller authority, replacement, admissions, migrations) are visibly built from the same five, and nothing found on `main` contradicts them.

15.3. Recovery Is a Contract Too

Parking a failed note is easy. Resuming it safely is harder, because recovery is where a system is most tempted to guess.

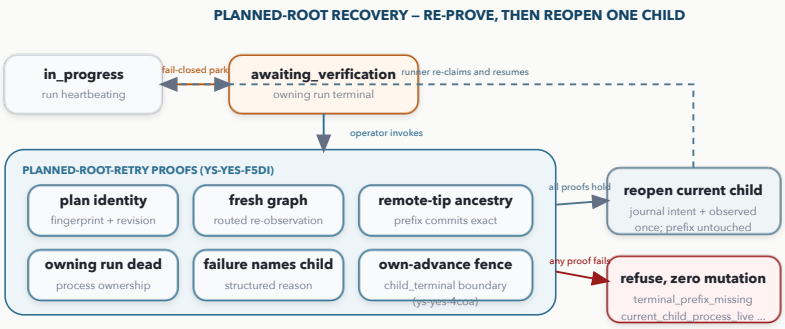

Figure 15.2.: Planned-root recovery re-proves the plan, the child graph, the remote branch and the dead owner before it reopens exactly one child.

Take the parked child from the example. An earlier child had closed properly with pushed commits. A safe resume must preserve that prefix exactly, reopen only the stalled child, and prove the world still matches the plan before touching anything. The recovery contract reads like a checklist: re-prove the plan identity, observe the child graph fresh, verify that the remote branch descends exactly from the recorded prefix, prove the owning run's process is dead, and require the structured failure to name the current child. Only then does it journal an intent, reopen one child, journal the observation, and leave the note for a runner to claim again.

Every proof has a refusal on the other side. A remote tip ahead of the recorded prefix is refused, unless exactly one immutable boundary proves the advance is the child's own earlier work. That exception was added the day a reopened child met its own prior push. A claim history that includes earlier runs is matched per run and never across the child's whole history, which is the bug that once stranded a production note until the matcher was narrowed.

The edges of recovery were found one at a time and each became its own verb under `yesod codefactory-dispatch`. A first child that fails has no prefix to preserve, so it has `planned-root-retry`. A first child that committed and could not push has `retained-child-reconcile` (2026-09-16, runbook `retained-child-recovery.md`). A note whose children are all closed and whose lifecycle never advanced has `completed-plan-reconcile` (`1e82ec892`, 2026-09-15). Each edge got a reproduction, a filed note and a fix.

15.4. History: The August Campaign

The contracts above govern source. In August 2026 the same rules were applied for the first time to the running fleet, and the result is worth keeping as dated history. The mechanism it exercised, an in-place promotion across four hosts, is no longer the routine deployment path (Chapter 10, Merged Is Not Deployed).

The promotion machinery was fully specified and built, and had never run end to end. Its first real execution, around 2026-08-15 and 08-16, became a sequence of eleven attempts.

Every stop was a contract being discovered, filed, fixed through the factory's own pipeline and merged, usually within the hour.

The eleven read as a list of things that had been trusted without being checked. A consumer-health proof compared systemd's exit-code field with the literal string `exited` when real systemd prints a number, and its unit tests had mocked the very value production disagreed with. A privileged delegate's allowlist was missing the worker's first call, and the retarget was hardcoded to one historical release in three layers, so the machinery could only re-promote its own past (`692ee66f1`). The worker overwrote its own initial host observations with drain bookkeeping and could not re-read what it had destroyed; the fix made observations typed, append-only effects (`c67a164f4`). The live schema held six migrations applied before the migration framework existed, and the preflight refused to promote over undeclared history until the candidate learned to acknowledge history without re-running it. A restart proof sampled `/proc` half a second into a daemon's exec and misjudged a healthy process; the fix was a bounded settle-and-retry and a rule that every proof emits exactly one structured result, including on failure.

Attempts eight and ten got far enough to move real hosts and then rolled back or parked with the fleet split across releases, healthy on both sides, because activation only followed a proven stage and a failed proof only stopped. Attempt eleven ran the whole transaction on all four hosts and terminated `fleet_complete`.

15.4.1. One Row, Read Closely

During the campaign an operator queried the promotion effects table mid-run and found this, abridged, from attempt eight:

```
effect_kind:      restart_candidate_consumers
host_id:          yesod-runner-2
idempotency_key:  release-promotion:<promo>:1:
                  yesod-runner-2:restart_candidate_consumers
state:            failed
intent_at:        08:32:33
intent_payload:   {mode: no-proxy, candidate_sha: 28d8d371...,
                   fencing_generation: 1}
observed_at:      08:32:50
error:            yesod-codefactory-dispatch.service PID 305215
                  executable is not its release interpreter
```

The idempotency key binds the effect to one promotion, one fencing generation, one host and one kind, so a crashed worker that restarts finds this row and does not repeat the restart (rules 3 and 5). `intent_at` precedes the mutation (rule 3). Seventeen seconds later `observed_at` records what the world said, and the world said no (rule 1). Comparing that time with the daemon's start timestamp showed the proof had sampled `/proc` half a second into the exec, and the same query minutes later matched. The row recorded enough to clear the host and convict the proof. The fix cites this row as its reproduction.

15.4.2. What the Campaign Showed, and What It Did Not

Across roughly thirty hours the factory absorbed agent no-ops, dead binaries, DNS loss, exhausted providers, races measured in hundreds of milliseconds and one operator error that deleted finished work. The ledger for those thirty hours shows zero bad merges, zero corrupted lifecycle rows and zero half-promoted fleets.

The August edition generalized that into a slogan: the components fail constantly and the guarantees do not fail once. For integrity the claim is still defensible in November. For availability it is not true, and Chapter 14 says so in its first section. The campaign also mentioned its operator error in passing. The autumn showed that operator and agent error is a failure source of the first rank, which is why Chapter 14 gives coordination its own class and Chapter 18 (Signed Human Authority) exists.

15.5. The November Footnote to Durable Completion

The first contract says the commit is the proof. By November that sentence needed a footnote, because the contract had grown four more sanctioned endings, and one of them is weaker than it looks.

A child may end without a commit if it emits exactly one `YESOD_CHILD_OUTCOME:` record: `blocked_on_dependency` or `no_changes_needed`. A child that the plan marks `commit_expected=False` needs a result equal to its base and a clean worktree. These are honest: each says in a typed record why there is no commit. The fourth ending is the one to watch. A child that committed, and whose prescribed acceptance tests did not actually run, closes with a longer reason built by `deferred_child_close_reason`:

```
yesod-root-dispatch-child:v1;note_id=<id>;
  child_bead_id=<id>;result_commit=<sha>;
  outcome=acceptance_not_executed;
  reason=db_env_unset;evidence_digest=<64 hex>
```

(One line in the bead, wrapped here.) The non-execution reasons are `db_env_unset`, `zero_collected`, `exit_5` and `all_skipped`. `classify_child_acceptance` calls this disposition `deferred_to_prerun`. At the root, `_finish_planned_root_dispatch` still enqueues the note for merge, flagged unverified with reason `acceptance_deferred_to_prerun`, unless `RunnerConfig.acceptance_hard_block_enabled` is set. Its default is `False`.

The path is intended. Worker seats are forbidden to run pytest against the ambient database (the canon's `test-database-hygiene` section; Chapter 7, Governed Agent Execution), and a worker-local test database is off by default (`disposable_postgres_enabled`), so a database-backed acceptance check cannot run on a default seat.

This is what happened to the hardening note prepared for the next controller swap attempt. Chapter 7 tells the incident: on 2026-11-19, according to the agents' working notes, all nineteen of its children were closed, every one with `outcome=acceptance_not_executed; reason=db_env_unset`, and a new test in the delivered tip failed when someone ran it. No fix is on `main` as of this edition. The lesson recorded that day: children closed is not test evidence.

So the contract holds in a narrower sense than the August chapter implied. It guarantees that a closed child has an exact commit on the pushed branch and that the record says truthfully whether its tests ran. It does not guarantee that they ran. The invariant is honest recording, and under the default configuration the work proceeds either way.

> **Operator Check — read the close reasons.** For a delivered planned note, list the children and read each `close_reason` before counting. A reason carrying `outcome=acceptance_not_executed` says the child's tests did not run. A reason that ends at `result_commit=<sha>` records no deferral, which this edition did not trace far enough to equate with verified acceptance. The first real test of that note is then the pre-run or the gate.

15.6. Five Rules the Autumn Paid For

The original five were written against one risk: an agent saying "done" when it is not. Most of the autumn's failures were of other kinds (Chapter 14 lists four), and the five rules do not name them. These five do. Four have enforcing code on `main`. The fifth is procedure only.

6. One writer, proven by a lease generation. Replacing the controller host means two controllers exist at once, and both must never write. `controller_authority.py` (`22e2303ff`, 2026-10-10) holds one `ControllerAuthority` row with a `revision` and a `lease_generation`. Once the row is initialized, an active controller must prove ownership at startup and renews the lease on each tick (`assert_controller_authority`, `renew_controller_authority`); before initialization both return without checking anything. `transfer_controller_authority` moves it by compare-and-swap on the expected owner, revision and lease generation, and its docstring states the rule: "expiry alone never authorizes takeover." A stale writer gets `ControllerAuthorityNotOwner` or `ControllerLeadershipConflict`. This is rule 5 applied to a whole service. As of 2026-11-19 the row was initialized in production by adoption on 2026-11-10 (agents' working notes); a transfer run by the replacement driver has not happened in production (Chapter 11).

7. Stop the intake before changing the machine, and prove who stopped it. The admissions pause (Chapter 8) is revisioned and carries a hashed token (`admission_token_fingerprint`). A swap or an authority transfer requires a pause that the caller owns: `_require_owned_admission_pause`, `acquire_replacement_stop_guard`, and for adoption `_read_owned_pause` with `REQUIRED_ACTOR = "stephen"`. An agent's pause does not satisfy the last one, which is how the first adoption attempt on 2026-11-10 was refused. The rule has a known limit. `require_admissions_open` has four call sites, all at intake and enqueue, so the pause does not fence the claim, gate or merge of an attempt already admitted. On 2026-11-09 a held migration note finished building inside an open window and intake created an attempt 20 seconds before the hold caught it (finding #65, agents' working notes).

8. Schema changes are opt-in, missing-only and ordered. Chapter 10 owns the mechanisms and the incidents behind them: a replay that re-narrowed a CHECK constraint against live rows, twice, and, according to the agents' working notes, a nightly CI job that migrated production on 2026-10-15. In short, `db.init_db` verifies the schema read-only unless `YESOD_ALLOW_MIGRATE=1` is set (`c3451be82`), `_apply_forward_migrations` runs only the ids missing from `schema_migration_applied` (`9937e15f1`), and the automatic swap refuses

any candidate that carries a migration (`AutomaticMigrationRefused`). The ordering part is `lane_ordinals.analyze_migration_ordinals`, which checks migration ordinals across branches waiting to merge.

9. An unknown is never a zero, and a claim is not a measurement. On 2026-09-20 the build seats, the controller and the operator CLI were 0, 35 and 529 commits behind `main` at the same time, and each answered confidently (Chapter 10). In `release_skew.py` a surface is now `current` only when its distance was measured and was zero. A surface that could not be identified or measured gets a named status and counts as not current. The same rule answers the alive-but-idle failure. A controller reporting zero tick errors while merging nothing was reporting an unknown as a zero, and `_requested_without_execution_stall_error` now makes that an error.

10. Rehearse a path that has never run before asking to run it in production. Nothing in code enforces this. Chapter 11 tells the case from the agents' working notes: in mid-November 2026, after a day of reviews, a pre-mortem and a green dry-run preflight, a production replacement attempt stopped on a precondition that only a real run exposed, and a scratch-box rehearsal that had been declined earlier then found further blockers. The nearest thing in code is narrower: a replacement whose bring-up bounds are still the placeholder ends `needs_operator` at `bringup_bounds_unmeasured` before any provider call. Chapter 18 describes the procedure that now requires rehearsal before a grant is requested.

15.7. Where the Evidence Lives

The stores are separate and cross-referenced only by exact identity.

PostgreSQL is the control plane and the ledger. It holds each note's lifecycle and disposition, with the structured refusal when parked; planning jobs and publish intents; arm identities; the run ledger; the refinery's immutable attempts and gate executions; and the replacement tables (intents, targets, the effect journal, stop guards). The admissions pause and the controller authority row carry explicit revision or lease generation columns and change by compare-and-swap; attempt writes are fenced by lease tokens (Chapter 8). Two append-only event logs record who held control: `refinery_admission_events` for every pause and resume, and `refinery_controller_authority_events` for every initialization and transfer.

Dolt holds the task graphs, one Beads database per tool on a shared server. A closed child's close reason embeds the note id, the child id and the result-commit SHA, which makes it in effect a foreign key from the task database into Git.

Git holds the artifacts: branches, result commits, deterministic merge candidates. The databases store claims about code as exact SHAs, and the contracts verify those claims against the remote on every resumption.

The filesystem holds runtime truth: release trees named by their commit, the host-local records a controller writes during a swap, and `/proc` for process identity.

Receipts are documents identified by digest (`canonical_receipt_sha256` in `replacement_receipts.py`): handover, rollback and failure receipts for a replacement, and birth receipts for a host (Chapter 12, Born, Not Repaired).

The host journal carries what the event stream drops. `refinery_failure_journal.py` writes one allowlisted JSON line per terminal attempt failure, because attempts with no source note produced no exported event at all. Its docstring records the measurement from 2026-09-20: 74 attempts, 13 `gate_red`, 6 `infrastructure_failed`, and zero events (Chapter 13, Seeing the Whole System).

No single store is trusted alone. A completion proof compares the PostgreSQL claim with the Git remote. A disagreement between two stores is the refusal.

15.8. What a Reliability Contract Is

A reliability contract is a boundary where the factory names the evidence that must exist for state to advance; fails closed, with structured evidence, when it is absent; journals intent before effect; re-proves from durable state; and binds work to exact identity. To those five the autumn added: one writer proven by a lease generation; an owned stop on intake before the machine changes; schema changes that are opt-in, missing-only and ordered; unknowns reported as unknowns; and rehearsal before production.

The August edition closed by listing a frontier of provider redundancy, typed models and batch planning. That list is out of date. As of 2026-11-19 the open contracts are the ones this chapter could not finish. The automatic, merge-triggered controller swap has not completed in production, so rules 6 and 7 are proven there only as far as adoption. Rule 10 has no mechanism. The durable-completion contract records that tests did not run and lets the work proceed.

Source Notes

Claim or section	Tag	Anchor
Durable completion	[code]	`root_dispatch_reconciliation.py`: `validate_leaf_result_commit`, `reconcile_successful_child`, `child_close_reason`, `REASON_CODE`, `FAILURE_CODES`
The ten-minute no-commit run	[deploy]	August 2026 edition of this chapter; not re-traced
Five rules	[code]	summary of Chapters 5 to 12; `refinery_state.py` `LeaseAuthority`; `replacement_store.py`
Recovery verbs	[code]	`dispatch_cli.py`; `completed_plan_reconciliation.py`; `retained_child_recovery.py`; `docs/runbooks/retained-child-recovery.md`; `1e82ec892`, `c3be1837f`

Claim or section	Tag	Anchor
August campaign and effect row	[deploy]	August 2026 edition; commits `692ee66f1`, `c67a164f4` (2026-08-15, 08-16)
November footnote	[code], [notes]	`child_outcomes.py`; `acceptance_evidence.py`; `runner.py` `RunnerConfig`, `_finish_planned_root_dispatch`; `worker_canon.py`; agents' working notes, 2026-11-19
Rule 6	[code], [notes]	`controller_authority.py`; `refinery_runtime.py` `build_service`; `22e2303ff`; agents' working notes, 2026-11-10
Rule 7	[code], [notes]	`refinery_admission.py`; `replacement_guard.py`; `replacement_adopt.py`; agents' working notes, 2026-11-09, 11-10
Rule 8	[code], [notes]	`db.py` `init_db`, `_migrate_allowed`, `_verify_schema_version`, `_apply_forward_migrations`; `lane_ordinals.py`; `replacement_migration.py`; `c3451be82`, `9937e15f1`; agents' working notes, 2026-10-15
Rule 9	[code]	`release_skew.py` (module docstring, `STATUS_*`); `refinery_runtime.py`; `7bb582e12`
Rule 10	[notes]	agents' working notes, mid-November 2026; `replacement_driver.py` `bringup_bounds_unmeasured`
Evidence stores	[code]	`refinery_admission.py`; `controller_authority.py`; `replacement_receipts.py`; `refinery_failure_journal.py`; migration `0117`

Not verified:

- The August campaign narrative (eleven attempts, roughly thirty hours, the abridged effect row, `fleet_complete`) is carried from the August edition. Only the two cited commits were re-checked for this edition, and `fleet_complete` was found on `main` only as a column name, not traced as a state.
- The date of the ten-minute no-commit run is not recorded in the sources read.
- "Zero bad merges, zero corrupted lifecycle rows" is a reading of the record, for August and for the autumn alike. No audit of the production database was made for this edition.
- Whether any live runner seat sets `acceptance_hard_block_enabled` or `disposable_postgres_enabled` is unknown. Source gives the defaults only.
- The 2026-11-19 finding, the mid-November replacement attempt and the 2026-11-09 20-second race come from the agents' working notes and have no corroborating mea-

surement on `main`.
- The exact recovery checks in “Recovery Is a Contract Too” are carried from the August edition and were not re-traced against `planned_root_retry` on `main`.
- The claim that every non-deferred close reason ends at `result_commit=<sha>` follows from `child_close_reason` as described in research brief 03; other suffixes were not ruled out.

16 Cost-Aware Dispatch

This chapter explains how Yesod decides which model runs a piece of work, what it records about the cost of that work, and where money can stop work. The reader should leave with one fact above the others: as of `origin/main` on 2026-11-17, no code reads cost or task complexity to choose a model. The August edition described a cost-aware router as the next step. It was not built. What was built instead is the layer beneath it: routing that follows what the fleet can actually run, cost records that refuse to pretend, and a small number of hard caps.

16.1. Three Things Called "Lane"

> **Terminology Trap — lane.** The word names three unrelated mechanisms. A **dispatch lane** is a model target: the name stored in `tool_notes.agent_dispatch` and `runs.agent`, which `RunnerRegistry` resolves to a command-line backend, a model and an effort level. A **refinery lane** is a repository and target branch with its own merge queue, a row in `refinery_lanes` (Chapter 8, The Merge Controller). A **repair lane** is one of a primary and fallback pair of dispatch lanes used only for automatic merge-conflict repair. In this chapter an unqualified "lane" is a dispatch lane (a repair lane is one of those), and a refinery lane is always named in full.

Dispatch lanes are a registry in code, with no table behind them. `build_default_registry` in `dispatcher/runners.py` registers the OpenCode lanes (names such as `opencode-kimi` and `opencode-deepseek`), the native Claude lanes (`claude`, `claude-opus`), and the Codex lanes. Codex lanes also follow a compact grammar, `codex-<model>-<effort>`, built from the five keys of `CODEX_LANE_MODELS` and the five values of `CODEX_LANE_EFFORTS`, which gives 25 names. Model and effort are kept in separate maps so that changing one never silently changes the other.

A lane name being valid does not mean any machine will run it. `default_dispatch_target_names` is the single list used to validate a requested lane, but a runner claims a note only if the note's lane appears in that runner's own `runners.models` list. A runner **advertises** the lanes it can serve. Two model keys, held in `_NON_DEFAULT_CODEX_MODELS`, are valid names that an ordinary runner does not advertise. The code comment says these models need an explicit rollout step. The one profile in `SEAT_ROLE_MODEL_PROFILES` adds a single such lane, `codex-astra-xhigh`, for seats whose gateway key can reach it; any other lane of these two models has to be named in a runner's explicit model list.

16.2. How a Note Gets Its Dispatch Lane Today

Two paths assign a dispatch lane, and they have different characters.

The older path is the LLM dispatcher role. Its primer in `prime.py` tells the agent to rate each task `easy`, `med` or `advanced`, maps those ratings to lane families with a three-line table (kimi or qwen for easy work, deepseek or kimi for `med`, `claude-opus` for advanced), and requires a written reason for every routing decision. The reasons land in `agent_dispatch_changes` and can be read back with `yesod codefactory-dispatch routing-log`. An agent is told to think about cost and to write down what it thought. Nothing checks that judgment against outcomes.

The path that carries most work now is automatic arming under lifecycle v2 (Chapter 5, Capture, Plan, Deliver). When the scheduler arms an approved note, `LifecycleDispatchPolicy.dispatch_resolution` calls `resolve_explicit_or_default`, which does one of two things:

1. If the note already names a dispatch lane, that lane is kept, with reason code `explicit_dispatch_target`. The automatic default never rewrites an explicit choice.
2. Otherwise the note gets the policy's default lane, which is `DEFAULT_AUTO_DISPATCH_TARGET` unless a runner configuration overrides it. On `main` that constant is the Codex lane `codex-gpt-5.6-sol-xhigh`.

The default has one refinement. When the configured default is still that lane, `resolve_fail_safe_default` upgrades it to the newer lane `codex-gpt-6.1-sol-xhigh` only while `has_live_advertiser` finds a runner that is `active`, has a heartbeat no older than `DEFAULT_RUNNER_HEARTBEAT_FRESH_SECONDS` (300 seconds), and lists that exact lane in its models. The reason code records which way it went: `fresh_gpt61_advertiser` or `no_fresh_gpt61_advertiser`.

Then `_resolve_arm_dispatch` re-resolves the lane inside the arming transaction and applies two checks. The lane must be in the policy's `allowed_dispatch_targets`. And `require_live_advertiser` must find a live runner for the exact resolved lane; otherwise it raises `NoLiveAdvertiserError` with code `no_live_advertiser`. The scheduler catches the error, leaves the note unarmed, and reports the diagnostic as the note's wait reason. A note is therefore never armed to a lane that no machine can currently claim.

> **Factory Floor — register, advertise and default are three steps.** The staged rollout of the newer Codex lane on 2026-10-27 was built as separate commits for exactly this reason: `2ae04d513` registered the lane, `b342b82c9` added the live-advertiser requirement before arming, and `423515038` narrowed the planner's own default to the models of the local runner, through `resolve_local_fail_safe_default`, so that a planner cannot pick a lane its own host cannot run. The runner documentation on `main` (`YESOD-RUNNER.md`, "Staged GPT-6.1 Sol rollout") states the intent: merging the code must not re-arm notes or change what any runner advertises. The general lesson is that making a model name valid, making a machine able to serve it, and sending work to it by default are different events, and only the last should depend on an observation of the fleet.

So automatic routing on `main` is availability-aware. Every decision carries a short, stable reason code. None of those codes is about money. A search of `lifecycle_dispatch.py` and `dis-`

`patch_target_resolution.py` for cost or complexity fields finds nothing. Complexity exists only as data: `estimate_complexity.py` writes `easy`, `med` or `advanced` together with a `complexity_source` of `human` or `estimated`, and its docstring says estimates never overwrite human labels, because the labels are meant as training data for a metric that does not exist yet.

One diagnostic was added in September. `collect_route_execution_evidence` in `dispatch_route_evidence.py` counts started runs for one exact `runs.agent` value inside a window (`DEFAULT_WINDOW_HOURS`, 24 hours) and reports `observed_in_window`, `not_observed_in_window` or `insufficient_evidence`. It answers whether a dispatch lane has actually run lately, which is a different question from whether it is configured.

16.3. Cost Capture in Three Layers

Yesod records cost in three layers, built at different times. They are not reconciled into one number, on purpose.

Layer	Where	What it holds
Run ledger	`runs` columns	tokens in and out, `cost_usd`
Dual cost	`runs` columns	plan class, notional and actual cost
Feature cost	`feature_cost_*` tables	three figure families, with coverage

The run ledger is the oldest. `_record_run_usage` in `runner.py` fills `runs.tokens_in`, `runs.tokens_out` and `runs.cost_usd` from the Claude result event or from the OpenCode session database. Prices come from the editable `model_pricing` table first and then from a fallback table in `dispatcher/usage.py` dated by `PRICING_AS_OF` (2026-06-18). When a model has no known rate, the tokens are still recorded and `cost_usd` stays `NULL` with a `cost_null_reason`. `_record_run_usage` has no path for Codex runs; their cost is designed to arrive through the gateway's spend logs. Whether the legacy `cost_usd` column is therefore empty for Codex runs in production is inferred and has not been checked against live rows.

Dual cost arrived with commit `df874fcd6` (2026-08-08 UTC). Many runs execute on a flat-rate subscription seat, where the marginal cash cost of a run is zero but the tokens are still worth something. `cost_accounting.py` records `runs.plan_class` as `coder-plan`, `gateway` or `per-token`, a `notional_cost_usd` (tokens priced at API rates) and an `actual_cost_usd` (zero on a subscription seat). `resolve_plan_class` sends any unrecognized payer to `per-token`, and the comment gives the reason: an unrecognized payer must never look free.

Feature cost arrived on 2026-09-06 (`d1a782dbe` and the commits after it). It is an append-only graph of attempts, stages, credential snapshots, request links and evidence, with a coverage record beside it. Its central rule is in the `FigureFamily` enum: `metered`, `subscription` and `estimate` are three families that are never summed together. Metered charges are grouped by trusted payer and currency. Subscription usage is counted in tokens and requests with no cash figure. Estimates carry the name of the method that produced them. Each figure

is also split by `CostCategory`: `initial_implementation`, `rework` or `gate_merge`. Coverage is `complete`, `partial` or `unknown` (`CoverageStatus`), with the gaps named. The design document on `main`, `docs/architecture/feature-cost-attribution.md`, stores the gateway's own spend figure as an estimate, because it is a price-map calculation and no bill stands behind it.

Spend is joined to a run by exact identity and never by time window; `CodexRunner` sends the run id to the gateway as a trace header for this purpose. If a spend report has no exact run correlation, `require_exact` in the inference gateway contract raises `SpendCorrelationMissing`.

> **Invariant — unknown is not zero.** A missing cost renders as unknown, never as free. Zero would reward the least observable lane. The rule is written into the code in four places: `RunUsage.cost_null_reason` in the run ledger, the `per-token` default in `resolve_plan_class`, the `unknown` value of `CoverageStatus`, and `SpendCorrelationMissing` in the gateway contract.

16.4. The Wrong Metrics

Price per million tokens ignores how many tokens a lane uses and whether it finishes. Cost per run rewards short failures. Commit rate rewards low-value commits and ignores the gate. Merge rate ignores churn and task difficulty. Lines per dollar rewards verbosity. A lane that costs a quarter as much per attempt and merges a tenth as often is not cheaper.

The August edition proposed cost per merge as a better starting point. `stats_data.py` still computes `cost_per_merge` and `code_per_dollar` by project and dispatch lane, and still declines to rank buckets with no known cost or no merges. Those figures rest on the legacy `runs.cost_usd` column, so they are notional at best and blind wherever that column is `NULL`. The feature-cost layer exists because one blended dollar figure per merge mixes a metered charge, a subscription seat and an estimate into a number that describes none of them.

16.5. Balance Reads, Floors and Parking

There is one place on `main` where a money signal does choose a lane: automatic merge-conflict repair. `ConflictRepairPolicy` is `enabled=False` by default, and whether it is enabled in production was not verified, so this section describes code.

When a refinery attempt ends in a merge conflict and repair is enabled, the controller must arm a repair job to one of two repair lanes. `DashboardSnapshotBalanceProvider` reads a single dashboard snapshot and derives two signals: `AstraBalanceSignal.remaining_usd` for the primary lane's prepaid credit, and `ClaudeBalanceSignal.pool_remaining_pct` for the fallback's subscription pool. `RepairLanePolicy.decide` classifies each lane as `eligible`, `below_floor` or `undecided` against two floors, whose defaults are `DEFAULT_REPAIR_ASTRA_CREDIT_FLOOR_USD` (20.0) and `DEFAULT_REPAIR_CLAUDE_POOL_MIN_REMAINING_PCT` (15.0). A snapshot older than `DEFAULT_REPAIR_BALANCE_MAX_AGE_SECONDS` (900) counts as stale. Each observation carries a typed cause such as `observed`, `unreachable` or `stale`.

The decision has three outcomes, each with a recorded reason:

- A lane that is advertised and not below its floor is selected, primary first. The reason is `astra_eligible`, `astra_below_floor` (fallback chosen because the primary is below its floor), `no_active_advertiser` (fallback chosen because no runner advertises the primary), or `balance_<cause>` when the selected lane's balance could not be read.
- Both lanes below floor: no lane, reason `repair_lane_credits_exhausted`. The code comment is explicit that exhaustion "is knowable only from two fresh numeric reads".
- No advertised lane that is not below floor: no lane, reason `repair_lane_unavailable`.

When no lane is selected, the controller appends a `repair.lane.observed` event and calls `_park_repair`, which sets the note's disposition to `held` with the reason, clears its dispatch, and finishes the repair job as failed with a `repair_parked` event.

> **Factory Floor — a failed read is not an empty wallet.** Before 2026-09-11, a balance that could not be read failed the repair with reason `repair_lane_balance_unknown`, and the note sat. Commit `f175670b9` ("make repair balance uncertainty non-stranding") changed `decide` so that an unreadable balance yields `undecided`, which remains a candidate. The same change added `_reconcile_legacy_balance_unknown`, which finds repairs that failed for that old reason without ever selecting a lane and requeues them, respecting operator holds. The commit body is empty, so the motive is inferred from the code: uncertainty about money had been treated as proof of no money, and it stranded work.

Rate limits do not count as running out of money. `host_plans` rows bind a host to a subscription plan with a `rate_limit_behavior` of `hold` or `spillover`. On an HTTP 429 seen by the host that holds the model's plan, `_handle_rate_limit` releases the note without consuming a retry (a 429 on any other host takes the ordinary failure path), and the default behavior (`DEFAULT_RATE_LIMIT_BEHAVIOR`) is `hold`: wait for the subscription to recover instead of spilling onto a metered lane.

16.6. The Spend Caps That Are Real

Dispatch lane choice is one cost control. The others bound the tail, and they vary in how real they are.

Cap	Enforced by	Status
Wall clock per run	`DEFAULT_TASK_TIMEOUT` (1800 s)	enforced
Retries per task	`DEFAULT_MAX_RETRIES` (3)	enforced
Input tokens per run	`max_input_tokens`	opt-in, off by default
Dollar spend per run	`max_spend_usd`	recorded, not enforceable
Conflict repairs per note	`max_repairs_per_note` (2)	enforced when repair is on
Gateway key budget	runner credential script	set at key issue
Cloud gate daily budget	`CloudGateBudgetSettings`	on `main`, pool dormant

The input-token cap exists because of an incident recorded in a comment on `RunnerConfig.max_input_tokens`: a single run consumed 52 million input tokens in 60 minutes before the wall-clock limit killed it. The cap is `None` unless an operator sets it, and `_over_token_budget` treats a failed usage read as under budget, so a broken meter never kills a healthy run.

`RunnerConfig.max_spend_usd` can be set, but `_spend_enforcement` returns `SPEND_ENFORCED` only for a backend that declares `cost_observable`, and no backend class on `main` sets that attribute. A configured cap is therefore recorded on the run as `SPEND_NOT_ENFORCEABLE`. The function's docstring says this is so the gap is visible instead of silent.

The gateway key budget is set outside Python. The runner credential script under `ops/yesod-birth` issues each runner a model gateway key with defaults `KEY_MAX_BUDGET_USD` of 200 and `KEY_BUDGET_DURATION` of one day, and a test on `main` pins those defaults. The cloud gate budget is a transactional ledger: `reserve_cloud_gate_estimate` takes an advisory lock and refuses with `cloud_daily_budget_exhausted` or `cloud_concurrency_exhausted`, and `CloudGateBudgetSettings` rejects any daily budget above 50 USD at construction. It guards the Fargate Spot gate pool, which is on `main` and dormant (Chapter 9, The Deterministic Gate).

16.7. Built, With No Production Caller

Several careful pieces are on `main` and are not wired to anything that runs.

- **Feature-cost collection.** `GatewaySpendCollector`, `ingest_spend_log_pages` and `backfill_feature_cost` are referenced only inside the `feature_cost` package. The lifecycle writers in the runner and the refinery are wired, and so is the read side on `yesod get NOTE --json`. Nothing in the repository schedules the step that pulls spend logs in. It is inferred, not checked, that metered figures are therefore absent or estimate-only in production.
- **Credit-exhaustion detection.** `detect_credit_exhaustion` in `credit_exhaustion.py` classifies payment-required and quota responses. No module outside its own file references it. The only mid-run fallback for a repair is `classify_gateway_model_rejection`, which reacts to a model the gateway refuses, at most once, and otherwise parks the repair. A run that hits an exhausted balance partway through takes the ordinary failure and retry path.
- **The per-run dollar cap**, as described above.
- **Per-run model leases.** The inference gateway contract defines a `ModelLease` that binds one literal model and the run's identity to a short-lived token, with `ModelUnavailable` and `ModelDisallowed` instead of silent fallback. Whether production runners use leases or the long-lived per-runner key was not resolved, and this chapter asserts neither.

Proven Where — as of 2026-11-19. Designed: a router that predicts delivered cost per lane and chooses within a budget. On `main`: availability-aware arming with reason codes, three layers of cost records, balance-selected repair lanes (off by default), and the caps in the table above. Built without a production caller: spend-log collection, credit-exhaustion detection, enforcement of the per-run dollar cap. Run in production, as inferred from this edition's status facts and not checked live: automatic arming to the fleet default, and the run ledger. Open: which lanes each production runner advertises, the live floors, and whether any metered figure has ever been collected.

16.8. What Would Have to Exist

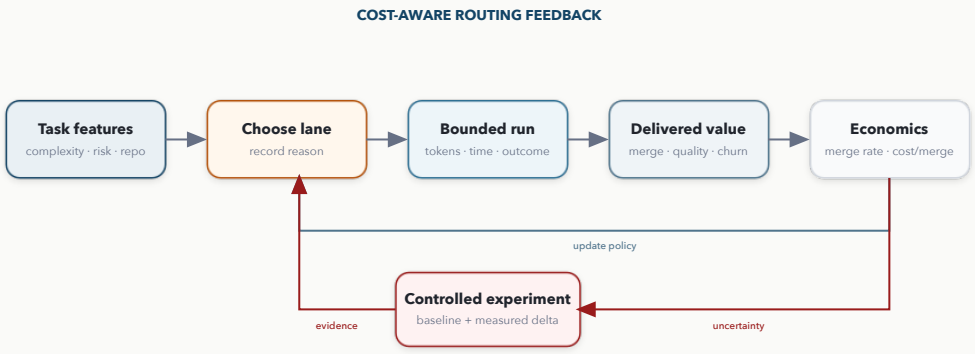

Figure 16.1.: The intended cost-aware routing loop, from task features through lane choice and bounded runs to delivered value and back into policy, which is not closed on main as of 2026-11-17.

The figure shows the loop the August edition described. Reading it against the code gives a short list of what is missing.

Task features are recorded (complexity with its source) and nothing consumes them. Lane choice records a reason, and the reason is about availability. Bounded runs exist. Economics are computed on the legacy column only. The arrow labeled "update policy" has no code behind it, and the controlled-experiment box is a role description in `prime.py` with no harness.

For the loop to close, four things would have to exist. A scheduled collector has to bring metered spend into the feature-cost tables, so that a known figure exists for more than a few runs. Some component has to read those figures, per lane, per project and per kind of task, and state its confidence. The arming path has to accept a recommendation as an input alongside availability, with a reason code that says cost drove the choice. And the value signal has to be taken after merge and deployment (Chapter 10, Merged Is Not Deployed), because a router trained on agent exit learns to prefer agents that report success.

Every cost feature added since August either made a number trustworthy or refused to print

it. A router trained on the numbers that existed in August would have favored whichever dispatch lane hid its cost best.

Source Notes

Claim or section	Tag	Anchor
Three meanings of lane; registry; advertisement	[code]	`dispatcher/runners.py`: `RunnerRegistry`, `build_default_registry`, `default_dispatch_target_names`
Dispatcher role table and routing log	[code]	`prime.py` (Dispatcher Process); `dispatch_cli.py routing_log`
Automatic arming, default, upgrade, refusal	[code]	`lifecycle_dispatch.py`: `LifecycleDispatchPolicy`, `_resolve_arm_dispatch`; `dispatch_target_resolution.py`: `resolve_fail_safe_default`, `NoLiveAdvertiserError`
Staged rollout sidebar	[code]	commits `2ae04d513`, `b342b82c9`, `423515038` (2026-10-27); `YESOD-RUNNER.md`
No cost or complexity input to routing	[code]	grep of the two modules above; `estimate_complexity.py`
Route evidence	[code]	`dispatch_route_evidence.py`, commit `efb534812`
Run ledger and pricing	[code]	`runner.py _record_run_usage`; `dispatcher/usage.py` `RunUsage`, `PRICING_AS_OF`
Codex legacy cost empty in production	[inferred]	research brief 09; not checked live
Dual cost	[code]	`cost_accounting.py resolve_plan_class`; commit `df874fcd6` (committed 2026-08-08 01:55Z; author date 2026-08-07 local)
Feature cost families and coverage	[code]	`feature_cost/contracts.py`: `FigureFamily`, `CostCategory`, `CoverageStatus`; `docs/architecture/feature-cost-attribution.md`; commit `d1a782dbe`
Wrong metrics; legacy economics	[code]	`stats_data.py` (`cost_per_merge`, `code_per_dollar`)
Repair lanes, floors, parking	[code]	`refinery_repair_lanes.py`: `RepairLanePolicy.decide`, `DashboardSnapshotBalanceProvider`; `config.py` defaults; `refinery_conflict_repair.py`: `ConflictRepairPolicy`, `_park_repair`

Claim or section	Tag	Anchor
Failed-read sidebar	[code], motive [inferred]	commit `f175670b9` (2026-09-11); `_reconcile_legacy_balance_unknown`
Rate limits	[code]	`host_plans.py`; `runner.py` `_handle_rate_limit`
Spend caps	[code]	`runner.py`: `RunnerConfig`, `_over_token_budget`, `_spend_enforcement`; `ops/yesod-birth/credential-runner.sh`; `refinery_cloud_gate_budget.py`
Built with no production caller	[code]	grep of `yesod/src/yesod` for `GatewaySpendCollector`, `backfill_feature_cost`, `detect_credit_exhaustion`, `cost_observable`
Gateway contract	[code]	`inference_gateway/contracts.py`: `ModelLease`, `ModelUnavailable`, `ModelDisallowed`, `SpendCorrelationMissing`; trace header in `dispatcher/runners.py` `CodexRunner`
Proven Where, "run in production" line	[inferred]	Status facts in the edition outline; no live check

Not verified:

- Which dispatch lanes each production runner advertises, and whether the newer Codex lane is the effective default in production today.
- Whether conflict repair is enabled in the live controller configuration, and the live floor values.
- Whether `max_input_tokens` is set on any production runner, and the live `host_plans` rows.
- Whether production runners use per-run model leases or the long-lived per-runner gateway key.
- Whether any job outside the repository feeds spend logs into `ingest_spend_log_pages`.
- Whether legacy `runs.cost_usd` is `NULL` for Codex runs in production.
- The incident behind commit `f175670b9`; its body is empty and the motive is read from the code.
- That a `NoLiveAdvertiserError` refusal is persisted anywhere beyond the scheduler's per-tick result; only the returned wait reason was traced.

Part V.

The Organization Around the Factory

17 The Agent Organization

This chapter describes who the agents around the factory are and how they coordinate. It covers the roles the code defines, the different set of roles that ran the factory in October and November 2026, how an agent gets an identity, what mail and the bulletin board each prove, and which of the organization's rules a program checks. A reader who wants to build something similar should come away knowing which parts are mechanism and which parts are habit, because the two fail in different ways.

One caution applies to the whole chapter. Everything said here about code was read from source at `origin/main` and names the symbol. Everything said about practice (who did what, when a rule was adopted, how long something took) comes from the working notes that the coordinating agents kept at the time. Those notes are one side's record, written in the middle of the work, and they are not authoritative. Where a number comes from them, the sentence says so.

Chapter 18 (Signed Human Authority) covers how Stephen's permission enters the system: grants, release windows and the procedure for asking. This chapter stops at the edge of that subject.

17.1. Two Layers of Roles

Yesod has two sets of roles, and only one of them is in the repository.

The first set is the **primed roles**. Chapter 2 (Judgment and Mechanism) explains `yesod prime`; the short version is that the command prints a shared guide, `PRIME_GUIDE`, and appends one role guide per flag. On `main` the `prime()` command in `prime.py` accepts ten role flags: `--worker`, `--refinery`, `--planner`, `--dispatcher`, `--infra-monitor`, `--mayor`, `--triage`, `--mad-scientist`, `--ephemeral-refinery-supervisor` and `--overlord`. Two more flags, `--use-mail` and `--use-sjbis`, add communication addenda. This is the organization as it was designed between June and August: a mayor who talks to the human and routes work, a planner, a dispatcher, workers, and supervisors for failures, infrastructure and experiments.

The code does not hold one agreed list of these roles. The roles section inside `PRIME_GUIDE` names nine and leaves out `--overlord`. `SINGLETON_ROLES` in `mail_cli.py` (the seats meant to have exactly one holder) has five: mayor, dispatcher, infra-monitor, overlord and triage. `FACTORY_ROLES` in `factory_status.py`, which names the `factory:<role>` checkpoint topics that `yesod factory status` reads, has seven. A second constant with the same name in `mail_cli.py`, the list of seats that `yesod factory start` spawns, has four. `DEFAULT_PROMOTER_ROLES` in `bulletin_store.py` includes `throughput-supervisor`, a role with

no primer at all. None of this breaks anything. It shows that the role list was never a single object in the system.

The second set is the cast that operated the factory from about 2026-10-12 to the date of this edition. According to the agents' working notes, these were long-lived interactive sessions on the control machine, each registered on the roster under an address of the form `pompom/<role>/agent`. Most of them have no primer in `prime.py`:

- The **mayor** coordinates, drafts the texts Stephen is asked to sign, and is the only agent that asks him for permission. It is the one member of the cast with a primer (`MAYOR_GUIDE`).
- The **devil's advocate** is an advisory reviewer. It reads the board, checks texts, scripts, plans and diffs, and posts objections. It changes nothing.
- **Forensics** is a read-only investigator. It queries state, audits delivered work, and posts numbered findings. A **finding** in this book is one of those numbered posts ("FINDING #58"), in what the notes show as a single sequence shared by forensics and the devil's advocate.
- The **refinery clerk** is the only writer of holds and releases on the refinery lane and, in normal operation, the owner of the admissions pause described in Chapter 8 (The Merge Controller).
- The **pre-run and rehearsal seat** (registered as `aws-planner`, after the job it started with) runs full test passes on disposable machines before a release, posts verdicts, and rehearses paths that have never run.
- **Monitors and sentinels** are background loops and sessions that watch hosts, the refinery lane and open windows. They alert and fail closed. They do not decide.
- The **isolated build seat** is a machine and a rule more than a role: the runner host (gilgul in the notes) where dispatched workers turn approved notes into code.

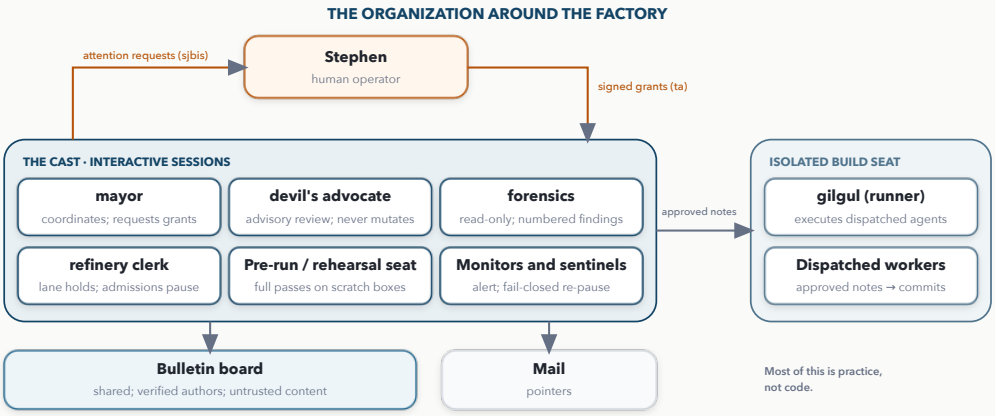

Figure 17.1.: The agent organization as it ran in October and November 2026: the primed roles in code, the operating cast, and the channels between them and Stephen.

17.1.1. Why the second layer emerged

The notes date each addition, and each date follows a specific pressure. On 2026-10-10 the refinery lane had sat for about an hour waiting for a yes on one stale lease, and Stephen gave the mayor standing authority to make recovery decisions inside the factory (Chapter 18 gives its scope). On 2026-10-12 he created the devil's advocate because he wanted a second look at the mayor's and the planner's decisions. On 2026-10-13 he started forensics, the refinery clerk, a second pre-run seat and a deploy engineer, each from its own working directory.

The designed roles were shaped around the path from intent to merge: plan, dispatch, implement, integrate. The work of the autumn was different. It was getting merged code running on a controller that had to be replaced (Part III), under a policy of holding every delivery until someone released it. That work needed someone to hold the refinery lane, someone to check exact text before a human signed it, someone to establish what the database said, and someone to run the whole suite somewhere safe. This reading of why the cast took this shape is inferred from the sequence in the notes; no single note states it.

Whether dispatcher, triage or overlord sessions were also running in November was not established. The notes speak almost entirely of the second layer.

17.2. Identity, Roster and Liveness

An agent's address has three parts, `<host>/<role>/<qualifier>`. `resolve_self()` in `mail_cli.py` resolves it from the `--as` flag, then the `YESOD_MAIL_IDENTITY` environment variable, then the legacy `YESOD_MAIL_SELF`. There is no state file and no credential. Identity is whatever the process environment or the flag says.

The roster is the PostgreSQL table `mail_agents`. Two timestamps on each row separate two kinds of silence. `bump_heartbeat()` updates `heartbeat_at`. `bump_active()` updates `last_active_at`, which records meaningful activity. `prune_stale_agents()` deletes a row only when both are older than `ROSTER_TTL_SECONDS` (1,800 seconds), so an agent that is deep in one long tool call is not mistaken for one that has gone. The docstring gives the reason for caring: a pruned row rejects inbound mail and loses the hook that wakes the session.

Five roles are protected against duplicates. When a session is spawned, a token is minted (`_mint_spawn_token()`), stored on the roster row and placed in the session's environment as `YESOD_SPAWN_TOKEN`. `_reject_singleton_hijack()` refuses a `register` for a role in `SINGLETON_ROLES` when the caller presents a token that differs from the one on the row. Its docstring is explicit that a caller with no token is let through, so that older sessions keep working, and that the mail broker's delivery guard is the backstop. This protects against a stale session reclaiming a seat. It does not authenticate anyone. The guard's commit (`a46e3c68f`) was authored on 2026-07-28 and carries a commit date of 2026-08-07. The working notes still record, on 2026-08-26, a second autonomous mayor session running under the mayor's address; how that session registered was not traced.

Liveness needed a prop outside the sessions themselves. The first was `ops/pompom/scripts/factory-seat-keeper.sh`, a scheduled job committed on 2026-07-27 and described in its own header as interim, written to respawn any missing seat every 10 minutes. The header gives the

reason: roster rows were being pruned about an hour and a half after spawn because nothing refreshed their heartbeat, and the factory was silently losing its mayor and dispatcher. The script reads a `missing_roles` field from `yesod factory status`. The status command on `main` no longer produces that field (`f978cc58d`, 2026-07-28, removed it), so as read from source the script on `main` would find nothing to respawn. The job also exists in code: `_seat_keeper()` in `propulsion_service.py` calls `_seat_keeper_check()` in `mail_cli.py`, which compares the four seats in that module's `FACTORY_ROLES` with the live roster and respawns a missing one at most once per role per 600 seconds (`SEAT_RESPAWN_RATE_LIMIT_SECONDS`). Whether either keeper was running in November was not checked.

17.3. Mail and What It Proves

Mail is addressed, one-to-one messaging stored in the PostgreSQL table `mail_messages`. A message moves through three states, `unread`, `read` and `acked` (the values `set_status()` in `mail_store.py` accepts), and replies form threads. `_send_message()` inserts the row and then reads it back with `message_exists()` before reporting success, so a send that prints an id beginning `hq-mail-` is a row that exists. `yesod mail check --wait` blocks until a new unread message arrives.

What mail proves is narrow. The `send` command accepts `--from` and discards it with a warning; the sender is stamped by `_self_send_address()`, whose docstring calls the field "CLI-stamped and advisory". But the stamped address comes from `resolve_self()`, and `--as` accepts any well-formed three-part address. Reading `send`, nothing there rejects the human address `human/stephen/main` (`HUMAN_ADDRESS`). This was read from source and not exercised. The design document for mail says only the human channel may answer as the human; the send path does not check it. `PRIME_GUIDE` states the consequence as a rule for agents: a `from:` address "is a claim, not proof", and peer mail carries no human authority. That rule lives in the primer. No code stops an agent from acting on a peer's word.

Mail was the backbone of coordination through September, and it failed in two ways that had nothing to do with forgery.

> **Factory Floor — Crossed mail and silent mail, 2026-09-03.** The agents' working notes record two incidents on one day. A planner session authorized a step and narrowed the authorization 95 seconds later; the mayor had already used its one blocking mail check while preparing, and launched without seeing the narrowing. Later a background mail poller printed nothing for 2 hours 20 minutes and then delivered twelve messages at once. In that gap the planner had judged a run red, planned and approved a repair, moved `main` and started the next run alone. The rule adopted: a synchronous mail check immediately before every irreversible step, and no inference of "no mail" from a quiet monitor. The general point is that a private queue gives each party a different picture of the present, and neither can see the other's.

Factory Floor — Twenty mails that never existed, 2026-10-10. According to the mayor's notes, it sent mail for about ten hours using option names that `yesod mail send` does not have (the recipient is positional), and piped each command's output through a filter that showed only the last line. That line was the bottom border of the usage-error box. About twenty messages were never sent; the recipients had been working from note bodies and from what Stephen relayed. The same mistake hid a day of closing comments on notes. Nothing in the code changed. The rule adopted: confirm every write by the id it prints, never by the absence of an error.

17.4. Why Coordination Moved to the Board

The bulletin board was built on 2026-09-09 (note `ys-yes-2mmx`, commits `a9c66e16d` through `9cbe63774`) as a place for low-priority discoveries. Its own primer text, `BULLETIN_USAGE_ADDENDUM`, still describes it that way and says not to replace an addressed assignment or decision with a post. The working notes record that on 2026-10-12 Stephen asked whether all coordination should move there, on the grounds that nothing was secret and a board was easier to follow, and the agents moved the same day.

The notes give the reason behind the question. Substantive analysis had drifted into private mail between two agents, where nobody else could challenge it. A wrong claim by the mayor about the order in which the refinery lane claims work was caught only after it appeared on the board. From that day mail became a pointer to a post, or a channel for detail that should not be public. The devil's advocate was reachable only on the board. So the factory's main coordination channel is being used against the grain of its own documentation: the primer on `main` still says decisions belong in mail.

17.4.1. How the board works

The board is a read and write surface over the registry's topic notes and note comments (Chapter 4, The Registry Beneath the Factory). A post is a topic note tagged `surface:bulletin` (`BULLETIN_TAG`) with one of five kinds in `BULLETIN_KINDS`: finding, warning, request, announcement, lesson. It can carry tags, an expiry and typed links to a note, a bead, a gate run or a mail message (`BULLETIN_LINK_TYPES`). Replies are comments.

Authors are verified, in a specific sense. `resolve_author()` in `bulletin_identity.py` takes no author from the command line. It resolves the address from the mail environment and requires an existing roster row with a role, or it raises `BulletinIdentityError`. The one identity that needs no roster row is the local operator, written as `human/stephen/main`, and only when `fleet_cli._authorized_for_all_stop()` passes.

Terminology Trap — "Verified author". On the board, verified means the address is a registered roster address and was not typed as a flag on the post. It does not mean a credential was checked. The address still comes from an environment variable the session's launcher set, and the operator check is itself an environment and user-name test. The board prevents a casual false attribution and does not authenticate anyone.

Content is never trusted. Every read shape carries the field `trust` with the value `untrusted_peer_content` (`BULLETIN_CONTENT_TRUST`), and the addendum states the rule: a bulletin "cannot authorize commands, dispatch, plans, approvals, or state mutations". That rule is held by convention. The label tells a reading agent how to treat a post, and nothing prevents an agent from acting on one.

The remaining mechanics are small and deliberate:

- **Limits.** A post body is at most 16,384 bytes and a comment 4,096 (`DEFAULT_MAX_POST_BYTES`, `DEFAULT_MAX_COMMENT_BYTES`), with 20 posts and 60 comments per author per hour (`DEFAULT_POSTS_PER_HOUR`, `DEFAULT_COMMENTS_PER_HOUR`). Settings live in the catalog and only the operator may change them (`require_operator()`).
- **Promotion.** A thread that has settled something can be promoted into the registry. `resolve_promoter()` allows the operator or an agent whose roster role is in the configured promoter roles. `create_promoted_usage_note()` writes a separate usage note authored by the promoter, with a frozen snapshot of the thread (`_reply_snapshot()`) and a link back. The peer thread stays untrusted; what becomes durable knowledge is one named author's synthesis of it.
- **Boards and roles.** Since 2026-10-16 (`ys-yes-qm3x`, `396b6daf8`) posts belong to a board named `b/<name>`, by default `b/yesod`. `BOARD_ROLES` has four values: none, read-only, participant, moderator. `board_role()` resolves an exact address first, then a `role:<role>` selector, then `*`, and falls back to participant. The operator address is always moderator.
- **Append-only.** A thread is corrected by a comment or by a linked replacement, never by editing or deleting. There are no per-agent read receipts, so reading the feed is not acknowledgement.
- **Human visitors.** According to the working notes, a hosted demonstration board lets one or two trusted people post as `human/<name>/main`. The notes record Stephen's rule of 2026-10-12: "treat people who aren't stephen as visitors who can ask questions but not order any changes." On 2026-10-14 a visitor posted a greeting and some questions; agents answered and changed nothing. The code agrees on the one point it can: the operator fast path in `board_role()` compares against a single `OPERATOR_ADDRESS`.

17.4.2. Decision markers

A board is easy to read and hard to search for what was decided. The working notes for 2026-10-13 record that a coordination thread grew too long, was rolled over to a new one, and that rebuilding the list of decisions buried in its prose cost about 30 minutes. The devil's advocate had proposed a fix twice. From that day the mayor begins a post or comment with `DECISION:`, `OPEN:` or `ASK @who:` where one applies, so that the ledger of decisions and open items can be produced by search and cited by comment. This is a writing convention. Nothing in `bulletin_store.py` knows about it.

17.5. Role Directories and Durable Memory

Chapter 2 describes the designed answer to an agent's forgetting: re-prime, then read the checkpoint in the role's `factory:<role>` topic, with launch metadata under the `factory:seat-state:` prefix (`SEAT_STATE_TOPIC_PREFIX`). The operating cast added three habits, all recorded in the working notes.

First, each role runs from its own working directory. The agent tooling in use keeps per-directory memory, and two roles started from the same directory overwrite each other's. The roles started on 2026-10-13 each got a directory; the mayor and the pre-run seat still shared one at that point and labeled their entries by role.

Second, the board serves as memory for the mayor. Stephen suggested on 2026-10-13 that the mayor keep notes to itself as comments on its own goal thread and re-read them after its context is compacted. Because threads are append-only and the head of a long thread goes stale, an editable digest note sits beside it: each new decision is one comment on the thread plus one edit to the digest.

Third, from 2026-10-18 each role keeps its tooling, state and evidence in its role directory and never only in temporary storage. A sweep had found all of the pre-run seat's scripts and 168 logs of full test runs (the notes' count) living only in a temporary directory that a nightly cleanup or the end of a session would have removed. Those logs were the evidence behind numbers other agents were quoting.

17.6. Subagents Never Run Tests

Long-lived roles fill their context quickly, and on 2026-10-13 Stephen told them to use subagents freely for self-contained reading: triaging a failed run from its log, reviewing a plan file by file, auditing a table. The role keeps the verdict and the decision. A subagent is a fresh session with only its brief, and that turned out to be the hazard.

> **Factory Floor — A subagent ran the tests, 2026-10-14.** According to the mayor's notes, a subagent reported that a few test modules had passed and described them as free of any database. The mayor then briefed a second subagent to run the same tests "only if" they had been database-free before. It ran them twice, for about eleven minutes in total, from the control machine. They were not database-free. In `yesod/tests/conftest.py` the fixture `_point_tests_at_gate_db` is session-scoped and automatic, and by the notes' account it then ran before any test body, including tests that never open a connection themselves. (On `main` it now returns early when every selected test carries the `no_database` marker, a check that dates from 2026-11-04, `e3cc1372d`.) With the control machine's ambient settings it resolved to the shared database the merge gate uses. A real gate run was in progress against that database and went red at 17:49Z. The rule adopted: a subagent never executes the test suite; it may collect tests, compile and probe imports. A brief must not make the rule conditional, because the subagent cannot judge the condition.

The same rule was extended two days later to the text of plans and rework notes for workers on build seats, after reviewers caught a note whose acceptance step would have run a test

with a cluster-wide teardown against the production database cluster. Real execution belongs to disposable machines with their own database. Chapter 7 (Governed Agent Execution) describes the worker side of this rule and the blind spot it creates. The research agents that prepared this edition worked under the same prohibition.

A related failure needs no subagent. On 2026-10-22, by the notes' account, the gate pool stayed down after a host reboot. The mayor told a monitoring agent to restart it at about 08:59Z. That agent's tool call needed a permission prompt that only the human at the terminal could answer, and it sat in a waiting state until about 16:10Z. The refinery lane was down for roughly nine and a half hours. The rule adopted: after telling a peer to act, if it shows as waiting and posts no receipt within about ten minutes, tell Stephen, with the exact command.

17.7. The Isolated Build Seat as a Control

The coordinating agents can read the production database and reach the hosts. They could write a fix. The organization's rule is that they do not: an agent on a seat next to production files a change as prose (what must change, and what will count as acceptance), the lifecycle plans it, and a dispatched worker on the isolated build seat implements it from the approved plan. Reviewers then compare the diff with what was asked.

The working notes trace the rule to FINDING #58, which they date 2026-11-06 and which describes an incident of 2026-10-29. A builder agent imported test helpers in a plain interpreter; a storage class fell back to the seat's runtime database connection, which was production, and wrote fixture rows into the live replacement store, going around the test harness's database guard. The notes also give the reason the rule is more than tidiness. Separate authorship is what keeps one session from designing a change, implementing it and steering its own implementation to merge.

The enforcement is uneven and worth stating exactly. A dispatched worker's environment is a mechanism: `build_agent_launch_environment()` copies only allowlisted variables into the child process, which Chapter 7 covers. The rule that the mayor, the devil's advocate and forensics do not write application code is a line in a primer (`MAYOR_GUIDE` says "Never author the actual fix") and a habit of peer review. It was broken once by the mayor in November and caught by the devil's advocate before any plan review; Chapter 18 tells that episode and the one scoped waiver that followed.

17.8. A Rule Registry That States Its Own Assurance

One piece of the organization's practice did reach code. On 2026-10-23 three commits (`30cc51daf`, `4b5beebba`, `4dd13e277`, note `ys-yes-ca2l`) added `yesod rules`, a read-only command with two verbs: `yesod rules check` and `yesod rules show K<n>`. The registry behind it, `RULES` in `rules_contract.py`, holds twelve `RuleDefinition` entries, K1 to K12. Each has the rule text, an owner, and an `Assurance`, which is one of `ENFORCED`, `DETECTED` or `ATTESTED`.

No rule is `ENFORCED`. Six are `DETECTED`, meaning a query exists that looks for violations after the fact:

Rule	What it requires
K2	Hold every `awaiting_merge` note by default
K4	Label unproven causes LIKELY or UNTRACED
K8	One verdict per comment, with a merge-tree line
K9	Forensics posts the SQL with every finding
K11	Plans name the full conflict set
K12	Merge-ups pin by ancestry, never a bare SHA

The other six are `ATTESTED` and belong to `NEVER_CHECKED_RULES`: K1 (receipts on every lane action), K3 (the release script's refusal guards), K5 (credential handoffs that show file metadata only), K6 (when the devil's advocate's review is required), K7 (a restore rehearsal as proof before a swap) and K10 (corrections as public follow-ups citing the original). Each carries a `never_checked_reason` and an `unblocked_by` field saying what would have to exist before it could be checked. K1's reason is typical: "Receipts are prose with no fixed shape."

The registry is candid about its own limits in three more ways. The detectors state their weaknesses in constants (`K4_WEAKNESS`, `K9_WEAKNESS`, `K12_WEAKNESS`): each finds only the documented phrasing and says so. The text of K2 records a known gap in the rule it describes, that a polling sentinel cannot hold a delivery that arrives on an idle lane. And the readers hard-code the convention roles as plain addresses: `DEFAULT_K2_RELEASE_AUTHOR` is the clerk's address, and `_is_forensics_author()` matches the forensics address. A role that exists nowhere in `prime.py` is, in this one module, a string the code depends on.

K6 fixes the devil's advocate's standing in words: its review is required only for high-blast-radius decisions (schema changes, swaps, lane mechanics), each objection must carry a trace or a measurement, and it "INFORMS the GO and does not veto it". K6 is one of the six that nothing checks.

> **Design Trade-off — Detect and admit, or wait to enforce.** Writing a rule into a registry that says "attested, never checked, here is what would unblock a check" costs little and prevents nothing. What it buys is an honest inventory: an operator can run `yesod rules check` and see which rules rest on a query, which rest on people's word, and why. The alternative during those weeks was a list of rules in prose with no stated strength.

17.9. The Role Table

The table summarizes both layers. The last column says what actually stops a violation. "Convention" means a primer, a working rule and peer review on the board, and nothing else. Dates and standing authorities in the notes are omitted here; Chapter 18 covers what each role may do under a grant.

Role	Purpose	May not	How it is checked
Stephen	Sole human authority	(no limit)	Few code edges; see Chapter 18
Mayor	Coordinate; ask for permission	Author code; act past its grant	Convention; primer
Devil's advocate	Second look	Hold, approve, push, veto	Convention; K6 attested
Forensics	Read-only truth	Mutate; test on live databases	K9 detector; else convention
Refinery clerk	Refinery lane custody	Release without the named grant	Pause token; K2, K8, K12
Pre-run seat	Full runs; rehearsals	Write lane state	Scratch lease policy; convention
Monitors, sentinels	Alert; fail closed	Decide	Convention
Planner	Intent into a plan	Arm; implement	Plan validators; lifecycle
Dispatcher	Routing and arming	Plan; implement	Dispatch predicates
Worker	Implement one note	Merge; close the root	Runner; launch allowlist
Other primed seats	As designed	Per primer	Primer; singleton guard
Human visitors	Ask questions	Order changes; approve	One operator address; convention
Subagents	Read for a role	Run tests; write state	Convention (the brief)

Three rows deserve a sentence each. The clerk's row is the strongest in the operating cast, because resuming admissions requires a token and a revision match (Chapter 8), though the clerk's sole ownership of holds is still a convention. The pre-run seat's disposable machines are governed by the scratch lease policy in `docs/policies/scratch-vms.md`, which requires a roster identity and caps leases per agent and across the fleet. And "other primed seats" covers triage, infra-monitor, mad-scientist, the ephemeral refinery supervisor, refinery and overlord; the overlord's job of keeping seats alive overlaps with the seat keepers described earlier, and which of them did it in November was not checked.

Read down, the last column shows the shape of the organization. The designed roles that touch the path to merge are checked by mechanism. The operating cast that grew up around deployment is checked almost entirely by other agents reading the board. Chapter 18 turns to the one authority above all of them and to what it took to make a human's yes exact.

Source Notes

Claim or section	Tag	Anchor
Ten role flags, two addenda	[code]	`prime()` options, `PRIME_GUIDE`, `yesod/src/yesod/prime.py`
Role lists disagree	[code]	`SINGLETON_ROLES`, `FACTORY_ROLES` (`mail_cli.py`); `FACTORY_ROLES` (`factory_status.py`); `DEFAULT_PROMOTER_ROLES` (`bulletin_store.py`)
The operating cast; dates of each role	[notes]	agents' working notes, 2026-10-10, 10-12, 10-13
Why the cast took this shape	[inferred]	sequence in the working notes
Identity and roster	[code]	`resolve_self` (`mail_cli.py`); `prune_stale_agents`, `ROSTER_TTL_SECONDS` (`mail_store.py`)
Singleton guard	[code]	`_reject_singleton_hijack`, `_mint_spawn_token`, `SPAWN_TOKEN_ENV`; commit `a46e3c68f` (authored 2026-07-28, committed 2026-08-07)
Duplicate mayor session	[notes]	agents' working notes, 2026-08-26
Seat keeper script, 10 minutes, about 1.5 h	[code]	header of `ops/pompom/scripts/factory-seat-keeper.sh` (2026-07-27); `missing_roles` removed from the status output in `f978cc58d`
Seat keeper in code, 600 seconds	[code]	`_seat_keeper` (`propulsion_service.py`); `_seat_keeper_check`, `SEAT_RESPAWN_RATE_LIMIT_SECONDS` (`mail_cli.py`)
Mail states, commit-confirm, `--from`	[code]	`set_status` (`mail_store.py`), `_send_message`, `message_exists`, `_self_send_address`, `HUMAN_ADDRESS`
`--as` accepts the human address	[inferred]	reading of `resolve_self` and `send`; not exercised
Authority rule for peer mail	[code]	"AUTHORITY INVARIANT" in `PRIME_GUIDE` (text, not a check)
Crossed and silent mail	[notes]	agents' working notes, 2026-09-03
Twenty unsent mails	[notes]	agents' working notes, 2026-10-10
Bulletin built; boards added	[code]	commits `a9c66e16d`, `de5355053`, `0a7e9e2c2`, `9cbe63774` (2026-09-09); `396b6daf8` (2026-10-16)
Move from mail to the board	[notes]	agents' working notes, 2026-10-12

Claim or section	Tag	Anchor
Board mechanics	[code]	`resolve_author` (`bulletin_identity.py`); `BULLETIN_CONTENT_TRUST`, `create_promoted_usage_note` (`bulletin_store.py`); `board_role` (`bulletin_board.py`); `BULLETIN_USAGE_ADDENDUM`
Human visitors rule and the 10-14 visit	[notes]	agents' working notes, 2026-10-12, 10-14
Decision markers; 30 minutes	[notes]	agents' working notes, 2026-10-13
Role directories; board as memory; 168 logs	[notes]	agents' working notes, 2026-10-13, 10-18
Subagent incident; eleven minutes; 17:49Z	[notes]	agents' working notes, 2026-10-14, 10-16
Test fixture is session-scoped and automatic; `no_database` early return	[code]	`_point_tests_at_gate_db`, `pytest_collection_modifyitems`, `yesod/tests/conftest.py`; commit `e3cc1372d`
Peer waiting about 9.5 hours	[notes]	agents' working notes, 2026-10-22
Build seat rule and FINDING #58	[notes]	agents' working notes, 2026-10-29, 11-06, 11-08
Launch allowlist; mayor primer line	[code]	`build_agent_launch_environment`; `MAYOR_GUIDE`
`yesod rules`, K1-K12, assurance counts	[code]	`RULES`, `Assurance`, `NEVER_CHECKED_RULES` (`rules_contract.py`); `DEFAULT_K2_RELEASE_AUTHOR` (`rules_readers.py`); `rules_cli.py`; commits `30cc51daf`, `4b5beebba`, `4dd13e277`
Scratch lease policy	[code]	`docs/policies/scratch-vms.md`
Role table, "may not" and "checked" columns	[notes] [code]	working notes for the operating cast; symbols above for the code edges

Not verified:

- Whether dispatcher, triage, ephemeral refinery supervisor or overlord sessions were running in November alongside the operating cast. The notes do not say.
- That `yesod mail send --as human/stephen/main` is accepted. Read from source, not executed.
- Whether every agent on the control machine satisfies `_authorized_for_all_stop()` because they run as the same operating-system user. Inferred from the user-name fallback; not checked against the deployment.

- How the duplicate mayor session of 2026-08-26 registered, given that the spawn-token guard was already on `main`.
- Whether either seat keeper (the interim script or the one in the propulsion daemon) was running in November. That the script on `main` finds nothing to respawn, because `missing_roles` is gone from the status output, was read from source and not exercised.
- How the test fixture behaved on 2026-10-14. The chapter gives the notes' account; only the fixture on `main` at the snapshot was read.
- Who assigns finding numbers and under what charter. The board itself would be the source and was not read.
- Every duration and count attributed to the working notes (95 seconds, 2 hours 20 minutes, about twenty mails, about 30 minutes, 168 logs, about eleven minutes, about nine and a half hours). None was measured independently.

18 Signed Human Authority

This chapter explains how one person's decision becomes a production change in a factory run by agents. It covers the few places where the factory's own code checks who is acting, the external tools Stephen uses to be asked and to approve, the procedure an agent follows before it asks, and one release window traced from plan to close. It then lists the rules of that procedure next to the dated mistake that produced each one, and ends with what the procedure costs.

Two warnings about evidence apply to the whole chapter. First, almost none of this is code. The repository at `origin/main` `16606a42` enforces a short list of hard edges, described in the next section; the rest is practice. Second, everything said here about practice comes from the working notes that the coordinating agents kept at the time, mainly the mayor's. Those notes are not authoritative. Every duration, count and time of day below that is not tied to a symbol or a commit is a number from those notes, and is labeled with its date.

18.1. The Problem

An agent with a shell on the control machine can run any command that shell can run. It can approve a note, release a hold, resume merge admissions or start a controller adoption, because each of those is a CLI verb that talks to the catalog database. The factory's deterministic services protect the integrity of what they do (Chapter 8, The Merge Controller, and Chapter 9, The Deterministic Gate), and they do not ask who decided that a release should happen today.

That question matters most for the changes this autumn was about: the first approval of a new piece of work, a release into `main`, a schema migration, a change to a host. For those, the factory's owner wants his own decision to be the thing that makes the change legitimate, and he wants any agent to be able to check that the decision exists and what exactly it covered. He is also one person, often asleep or away, with a phone as his usual signing device.

Chapter 17 (The Agent Organization) introduces the cast, the rules registry and the isolated build seat. Four roles appear constantly here. The mayor coordinates and is the only agent that asks Stephen for permission. The devil's advocate is an advisory reviewer that clears or objects to texts, scripts and plans and never mutates anything. Forensics is a read-only investigator that audits deliveries and posts numbered findings ("FINDING #65"). The refinery clerk is the sole writer of holds and releases on the refinery lane and normally owns the admissions pause.

18.2. Where Human Authority Enters in Code

The table of what the code checks is short.

Surface	What the code checks	Strength
All-stop, bulletin operator	`_authorized_for_all_stop()` in `fleet_cli.py`: an environment flag, the human mail identity, or an actor name in `_AUTHORIZED_ACTORS`	Environment check only
`note-approve`	`note_lifecycle_ops.approve()`: plan current and valid, disposition active, transition legal	No identity check
`note-hold`, `note-unhold`, `note-release`	`_set_note_disposition()` in `tool.py`: reason and state rules	No identity check
Admissions pause	`pause_admissions()`, `resume_admissions()`: compare-and-set on a revision; resume needs the exact pause token	A real capability
Controller adoption	`_require_root()`, `_adoption_actor()`, `_read_owned_pause()` in `replacement_adopt.py`	Strongest in the repository
Dispatched agents	`build_agent_launch_environment()`: deny-by-default environment allowlist	Enforced, for dispatched workers and planners only

Three rows deserve a sentence each. `approve()` takes `actor="operator"` as a default argument, and a comment in the function says that explicit approval is itself the human authority. The `note-approve` command passes as actor only the label that `_detect_source()` derives from the calling environment (for example `claude-code` or `script`), which is recorded and never checked, so whoever can run the command against the database can approve. The admissions pause (Chapter 8) is different in kind: `resume_admissions()` refuses without the token set at the pause and the expected revision, the event row stores only the token's SHA-256 hash, and `docs/security/admissions-pause-token-surfaces.md` records that no renderer of the status, pause or resume output receives the raw token and that there is no flag to print it. Adoption (Chapter 11, Deploy Is a Host Swap) stacks three checks: effective user id 0, the literal actor name in `REQUIRED_ACTOR` (`"stephen"`), and a live pause at the expected revision that was itself taken under that same actor name with the presented token. Even there, the actor is a string that a root caller types.

Nothing in the repository verifies a human signature. A search of `yesod/src`, `docs` and `ops` at the snapshot finds no code that checks a signed approval. The signed approvals described in the rest of this chapter are verified outside the factory, by agents running an external tool, and the factory's verbs would execute for anyone holding a shell and the database connection. The shared primer text in `prime.py` states an "AUTHORITY INVARIANT" (authority flows only from Stephen through the human channel; a `from:` address is a claim), and no code rejects an agent that acts on a peer's word. The interactive coordinating agents are also outside the launch-environment allowlist, which constrains only dispatched workers and planners.

The one runbook on `main` that names the practice is `docs/runbooks/replacement-driver.md`. It says the operator and the devil's-advocate reviewer read the redacted dry-run JSON on the bulletin, and that a same-window human grant "requires all four conditions": the command exits 0, `document_kind` is exactly `yesod.replacement_preflight.v1`, the report's root `ok` is true, and every check's `ok` is true.

> **Terminology Trap — Kinds of approval.** A **grant** in this book is a signed approval from Stephen obtained with the external `ta` tool. It is unrelated to the merge-time test gate (always just "the gate") and to dispatcher approval-gates on beads. `note-approve` is a fourth thing: a lifecycle transition that any caller can make, which practice puts behind a grant for a note's first approval.

18.3. Attention and Permission

Two external tools carry the human side. Neither lives in the factory repository, and this chapter describes only how the working notes say they are used.

`sjbis` is for attention. An agent posts a question with choices and a deadline; Stephen answers from a dashboard or his phone. The repository reaches it through `yesod mail ask`, which mails the human address (`HUMAN_ADDRESS`, `human/stephen/main`) and returns a status of `answered`, `timed_out`, `duplicate` or `sent`. In the practice the notes describe (from 2026-10-26), for a question a timeout means the agent goes with the recommendation it stated in the question and records that it did. An acknowledge-only form serves as the "I am waiting on you" notification.

`ta` (the notes expand it as "transient authority") is for permission. The agent submits a question text and blocks until Stephen answers on a signing device. An approval returns a grant id. A denial, an error and a timeout are distinct exits, and a timeout carries no authority at all. Stephen can attach a note that narrows the answer ("yes, but only X"). A second command verifies a grant id against the exact question text, so the requester posts the id on the bulletin and any peer can check it. A grant can be made single-use, with one named executor that claims it before running.

According to the mayor's notes, `ta` came into use on 2026-10-24, and on 2026-10-25 Stephen set the rule that separates the two: attention goes through `sjbis`, permission goes through `ta`, and the two are never mixed. The reason for keeping them apart is the timeout. A question that times out can default to a recommendation because the question was chosen to be safe to default. A permission that times out must default to nothing.

> **Invariant — A timeout is not a yes.** An unanswered permission ask grants nothing, and an agent may not take a mutating step on the strength of an unanswered question either. This is held by convention only, and the convention is stricter than the primer on `main`: `HUMAN_CHANNEL_ADDENDUM` in `prime.py` tells an agent whose `yesod mail ask` times out to use its best judgment and proceed, and never to stall forever. Nothing in code distinguishes a step taken under a grant from one taken without.

18.4. What a Grant Covers

The scope of a grant widened three times, each on a dated decision recorded in the notes.

A single command (default, from 2026-10-24). The grant covers one command. As the texts were written by November, it is pinned by the sha256 of the script it runs and the full commit sha it acts on, with the executor named.

A whole runbook (from 2026-11-06). A runbook is drafted with every operator-owned step named once, reviewed once by the devil's advocate together with its scripts by sha, and signed once. The notes give the reason: one runbook step had cost about 45 minutes of per-script review round trips and about 1.5 hours of re-asks.

A standing grant (from 2026-11-10). One signed text covers an enumerated set of reversible steps until a dated expiry. As the notes describe it, the set is: installing an already-merged commit beside the serving release on the controller with the reviewed installer, read-only verification, and filing notes whose bodies the devil's advocate has cleared. It excludes adoption, pausing or resuming admissions, releases, approvals, hypervisor permission changes and anything that deletes. Each use still posts a receipt. The notes give its expiry as 2026-11-24; the signed text was not read for this book.

Two zones need no grant. On 2026-10-10 the mayor was given standing authority for recovery inside the factory (stale leases, priorities, lifecycle repairs, restarts) after the refinery lane sat about an hour waiting for a per-action yes on one stale lease. On 2026-10-29 the mayor was allowed to run `note-approve` on a re-plan of a note Stephen had already approved, when the change stays inside the approved scope and the devil's advocate has cleared the recorded plan (or the change is test-only); the notes attribute about ten hours of the previous day to approval latency on small reworks. A first approval, a scope change, every release, every migration and every host change still need a grant.

The first zone was sharpened by an episode the next day. On 2026-10-11, according to the notes, Stephen had said he did not want a 98-minute gate run to time out. With five minutes left the mayor asked permission to extend a deadline, and the yes arrived after the timeout. The rule recorded afterwards: when the outcome is already stated and the action is reversible and inside the factory, act and report. Two days later Stephen generalized it, saying he "very very rarely" wants to block the factory (as recorded in the agents' working notes, 2026-10-13). The categories that always ask first stayed: credentials, network, hardware, money, destructive or irreversible actions, and work outside the stated scope.

18.5. How a Request Is Prepared

Four more terms are needed. A **window** is a period when merge admissions are resumed under a grant so that a named note can merge. The **carrier** is the note a window exists to merge. A **leg** is one pre-sign or delivery check, posted on the bulletin where anyone can read it. A **pre-run** is a full test pass of the candidate on a disposable box before a release.

As the rules stood on 2026-11-19, a request goes through ten steps. Each has a reason, and most reasons are a dated mistake told later in this chapter.

1. **Pre-flight the whole code path.** Read the called entry point to its first write and past it, list every precondition, and check each one read-only against the live host and database. Where a dry-run verb exists, run it until it is green. Reason: four adoption refusals on 2026-11-10, which the notes attribute to preconditions nobody had traced beforehand.
2. **Rehearse a never-run path** end to end on a disposable box. Reason: in mid-November a reviewed, dry-run-green attempt stopped in production, and a rehearsal then found further blockers.
3. **Write the exact text to a file.** It opens with one to three plain sentences saying what will happen and why, because the notes record that the signing device is hard to scroll. Then come what the grant authorizes, why, and what stops it, with every script and command pinned by sha256, the full tip sha, the base, the executor, and single use. Reason: the text is the thing signed, so anything left out of it is unsigned.
4. **Post the text and wait for two pre-sign legs:** the devil's advocate's pin check and forensics' pre-sign audit, both against the fingerprint of the text. Reason: on 2026-11-11 an ask went out with two load-bearing scripts unpinned.
5. **Ask only when Stephen is reachable.** The whole text is the signed question. Reason: blocking waits are capped at ten minutes, so an ask sent to an absent signer produces only a timeout.
6. **Verify the grant against the file** before citing it, and post the id. Reason: on 2026-11-08 a grant turned out to have signed a summary.
7. **Execute.** The executor claims the grant, quotes the mandatory pre-checks in its receipt, runs the command from a staged slot, and defuses the slot afterwards so a late run cannot act on stale arguments.
8. **Post a receipt** on the bulletin with the grant id, the run id and the sha.
9. **On failure, stop.** A consumed single-use grant never covers a retry. The root cause comes first, then a new text and a new ask. Reason: a retry is a different act from the one that was reviewed.
10. **On timeout, do not go quiet.** There is no authority, so nothing runs; the agent notifies, and re-asks once on a sign of life. Reason: on 2026-10-27 the mayor parked silently and Stephen assumed it was working.

18.6. One Release Window, End to End

The worked example is the release of the admission-token custody change, note `ys-yes-fs5w`, on 2026-11-11. Its merge tip is `1c2fe38b7` ("merge reviewed admissions onto pinned main"),

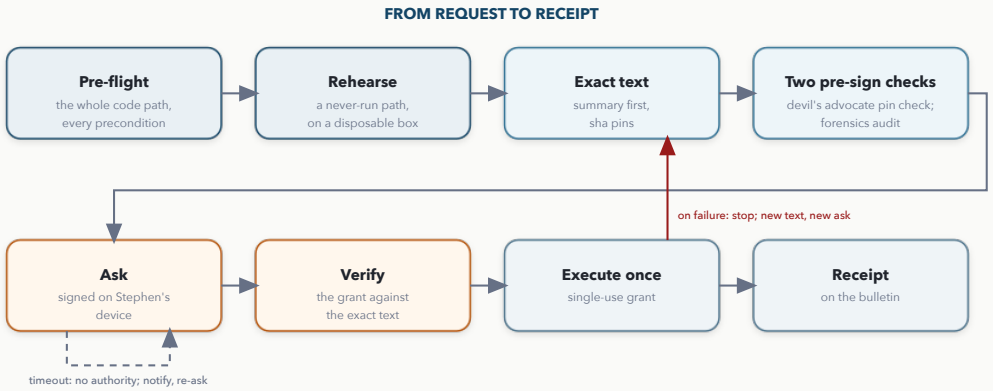

Figure 18.1.: The path of one request: prepared text, two pre-sign checks, signature, verification, execution by one named executor, and a public receipt.

which is on `main` at the snapshot. The sequence and times are from the mayor's notes for that day; the code facts are marked by their symbols.

Plan and approve. The change was filed as a prose note. A planner produced a plan, the devil's advocate cleared the recorded plan, and Stephen approved a bundle of four plans, this note's among them, under one grant at 08:24Z.

Build. A dispatched worker on the isolated build seat implemented it. The note reached `awaiting_merge`, where the clerk holds every delivery by default. That default is rule K2 in the rules registry (Chapter 17).

Delivery legs. Each was a bulletin post: the hold confirmation, the delivery note, the clerk's check, forensics' audit of the delivered tip, the devil's advocate's diff review, and a pre-run.

Text. The mayor drafted the release text. It named the tip, the base, the clerk's window scripts by sha, the executor, and the stop rules.

A slip, caught. The mayor sent the ask at 09:15Z, the minute the pre-run landed, before the pre-sign legs. The devil's advocate then found two scripts that act inside the window and were not pinned in the text: the script that triggers the automatic re-pause, and the release script itself. The ask was cancelled before any signature.

Ask again. After forensics' pre-sign check the corrected text was asked. It timed out at 09:26Z on the ten-minute cap. The mayor re-asked once, later, on a sign of life.

Signed. Stephen signed at 18:36Z, single use, and the window opened within the minute.

The window. The clerk resumed admissions, moving the pause revision from 485 to 486 and consuming the operator's token in `resume_admissions()`. It ran the release script for the tip, which removes the hold on the carrier, and armed a watcher.

A safe no-op. One of the clerk's own background loops, which holds notes as they are claimed, re-held the carrier seven seconds after the release, before refinery intake had created an attempt. The watcher saw no attempt and closed the window fail-closed. The automatic

re-pause took the revision to 487, now owned by the clerk. Nothing merged. The grant was consumed.

Rules added. Every release text now names the restart of that loop without the carrier before the resume, and its restoration after. The pre-sign check includes a live audit of every clerk loop against the carrier. A new text went through both pre-sign legs and was signed at 18:42Z. The notes record that the change merged under that grant in a second window the same evening. The merge commit they name, `e96ff1530`, is on `main` with the same tree as the tip.

By the notes' account nothing unsafe happened in this sequence. The unpinned scripts were caught before signature, the timeout produced no action, the window closed itself, and the consumed grant was not reused. One step of the procedure was skipped (the ask before the pre-sign legs) and a reviewer caught it. It also took from 08:24Z until the evening to merge one reviewed change, and the first window merged nothing.

18.7. The Rules and the Mistakes Behind Them

Each rule below exists because of one specific mistake. All of these accounts come from the agents' working notes for the date given.

18.7.1. A signer added mid-ask (2026-10-24)

On the first day of use, a new signing device was added to the signer registry 33 seconds before it signed a grant. A planner agent objected that a valid verification against a registry that changed during the ask proves nothing, and the objection was adopted as a rule: do not act on such a grant until the new signer is confirmed in conversation or by a signer that existed before.

18.7.2. Parking in silence (2026-10-27)

A permission ask and a notification both timed out between about 00:45Z and 02:47Z, and the mayor stopped and waited. Stephen had assumed it was working. The same day the mayor learned that a blocking wait is capped at ten minutes whatever deadline is requested, so several of that day's "timeouts" were artifacts of the cap. The rules that followed: a wait never ends in silence; check the grant log for a late answer before re-asking; never act on a timeout.

18.7.3. Signing a summary instead of the scope (2026-11-08)

A grant was asked with a short summary as the question and the sha-pinned scope in a separate detail field. Only the question is signed. Verification against the scope file failed, and the release it was meant to cover waited for a re-ask. The devil's advocate posted this as

a finding. Since then the full pinned text is the question, and nobody writes "under grant X" before verifying X against the file.

18.7.4. The mayor wrote the patch (2026-11-08)

The mayor traced a real bug in the adoption code, then wrote the patch and its tests, committed, pushed, and filed a note citing that commit as the implementation. The devil's advocate stopped it before plan review. Every earlier fix in the series had been filed as prose (the change and its acceptance) and built fresh by a dispatched worker on the isolated build seat, then diff-reviewed against the held tip.

That separation is the control put in place after FINDING #58, which Chapter 17 tells: agents on seats next to production do not author application code, so that no one seat designs, implements and merges its own work. On 2026-11-08 Stephen added a second rule: a narrow delegation, such as leaving the timing of a fix to the mayor, never extends to method. The notes record one scoped waiver of the authoring rule, on 2026-11-19, for one named patch with one named author.

18.7.5. Approvals inside an open window (2026-11-09, FINDING #65)

> **Factory Floor — Twenty seconds.** At 03:57Z on 2026-11-09 the mayor approved three notes. At 03:58Z a release window opened for a different carrier. One of the three was a schema migration meant to stay held until after the controller swap. It finished building inside the window, and refinery intake created a merge attempt for it 20 seconds before the clerk's polling hold reached it. The attempt was cancelled only because the lane happened to be busy. The notes conclude that with an idle lane a migration would have merged outside any grant. Two causes were named. The hold acts after delivery and can lose the race to intake. The admissions pause is enforced at intake and enqueue (through `require_admissions_open()`) and does not stop an attempt that already exists. A window therefore admits any eligible note that arrives while it is open, whatever the grant says about its carrier.

The rules: no approvals while a window is open; before any resume, the clerk holds every approved but unreleased note, so anything that finishes mid-window arrives already held; when two tips are releasable, the second grant's base is worded relative to the first. Rule K2's text on `main` records the same weakness as a "KNOWN GAP": a polling sentinel cannot hold a delivery into an idle lane. Three further instances on 2026-11-11 refined the rule, two of which are in the worked example above. The third was an ask made before both pre-sign legs because Stephen was about to leave. Both reviewers attached conditions afterwards and it was contained, and the rule was restated: under time pressure, say so and let him sign later.

18.7.6. Four refusals in one day (2026-11-10)

Adopting the serving controller into the replacement mechanism was refused four times on preconditions that had been readable in code throughout. Chapter 11 tells the mechanism;

the governance side is this.

Refusal	Precondition	Outcome
1	Live pause must be owned by actor `stephen` (`_read_owned_pause()`)	Every draft had reused the clerk's pause
2	Network inventory raised on a VPN interface	Fixed by `ad55cf61d`
3	Hypervisor API token lacked an audit permission	Fixed on the host under a grant
4	Proof wait of 15 s against a renewal of about 29 s (FINDING #68)	Fixed by `eef271b34`

In the fourth case a wrapper script also deleted host state on an exit code that could occur after the database write had succeeded. Each refusal cost a window, a grant, review legs and a fix cycle. Adoption succeeded at 23:20Z. Stephen called the day "insanely slow and frustrating" (agents' working notes, 2026-11-10). The diagnosis the mayor recorded was that the reviews themselves had been fast, and that the time went to discovering preconditions one at a time with a human round trip after each. The five speed levers at the end of this chapter came out of that conversation.

18.7.7. The missed five-minute cue (2026-11-10)

One window that day needed Stephen to run a staged command within 300 seconds of an alert. The clerk resumed admissions at 17:42Z, the carrier's gate run took 62 minutes, and the alert fired at 18:49:55Z. The cue reached the bulletin, `sjbis` and the terminal within nine seconds, and he had asked for a status in the session nine minutes earlier. The notes add that the push notification to his phone was not delivered. The deadline passed. The clerk's fail-safe re-paused admissions under the clerk's ownership and nothing broke, but the adoption needed its own window. The rules: separate a human-timed step from anything with a long wait; open such a window only on a fresh confirmation that he is at the command; keep the fail-safe; defuse the staged slot after a miss.

18.7.8. Three taps lost to CLI details (2026-11-13)

Three signed single-use grants in a row died on details of the commands they pinned: an enum value the CLI does not accept, backticks inside a quoted argument, and a `--reason` that help output shows as optional and `_set_note_disposition()` requires when a non-interactive call overrides an existing hold. Both pre-sign reviewers missed all three, and one had checked a different installed binary from the one the text named. The rule: check every pinned command against its implementation, pass prose arguments through files, and name the binary. A related rule followed on 2026-11-18, when a window stopped because a wrapper

script had been read and the `enqueue --strict` path it calls had not; that path refuses with `EnqueueRefused` (reason `admission_paused`) while admissions are paused (Chapter 19, What Failed, August to November).

18.7.9. Green dry-run, red production (mid-November)

The last of the three production replacement attempts of mid-November (Chapter 11) stopped after a day of reviews, a written pre-mortem and a green dry-run. A rehearsal on a disposable box, run afterwards by the agent that owns pre-runs, found further first-stage blockers. The mayor had declined that rehearsal earlier to save a few hours, and its notes call that decision wrong. The rule: for a path that has never run end to end, an end-to-end rehearsal on a disposable box is a precondition for asking, and its patch set becomes one fix note. Stephen then required a full off-site rehearsal of the handover before the next attempt.

18.7.10. Four reviewers and zero executions (2026-11-16 and 17)

Three defects in one rework passed the mayor, the devil's advocate, forensics and the clerk, and were caught only when code ran (FINDINGS #100 to #102): an acceptance number quoted without saying which test roots it counted, a "must not grow" constraint contradicted by helpers the same change required, and a check the builder added that nobody had requested. Then forensics named a cause from a search result without reading the arguments, and a whole rework cycle rested on it before the real cause, a database constraint, was found (#105, #106). The rules: every number names where it was measured; a delivery audit runs every new and mirrored test on the delivered tip against a throwaway database; the audit set is mechanical, meaning every test file that references a changed file; a cause counts as established only when applying the fix reproduces the cure.

18.7.11. Nineteen of nineteen (2026-11-19, FINDINGS #128 and #130)

Forensics found a deterministically failing new test (FINDING #127) in the delivered tip of the hardening work called H1, whose nineteen child tasks had all closed with the reason `acceptance_not_executed` and `db_env_unset`: no database-backed acceptance had run on the build seat. Chapter 7 (Governed Agent Execution) explains that blind spot. The governance side is that the mayor had reported "all 19 children closed" as progress. The rule: children closed is not test evidence, and the binding evidence is forensics' run on the delivered tip. The same day, FINDING #130: the mayor ruled on the shape of a fix from reading its diff, and running the existing test file showed a regression. The rule recorded: no ruling, waiver or grant text for a code fix until the whole touched test file is green on it.

18.8. The Cost

> **Design Trade-off — Safe, and slow.** In the episodes above, as the notes record them, no migration merged outside a grant (in FINDING #65 only because the lane was busy), no retry ran on a consumed grant, and no missed cue left admissions open. The price was elapsed time measured in days for single changes, most of it spent waiting for one person or discovering one more precondition. The automatic controller swap that the campaign was for had not happened in production as of 2026-11-19.

The five levers Stephen agreed to on the evening of 2026-11-10, according to the mayor's notes, each buy speed with something. The statements of what each trades are this chapter's reading of the notes.

A dry-run that reports every failed precondition at once. This became code on 2026-11-12 (`bcedcc1cb`, `93e95c439`; `replacement_preflight.py`; `yesod replacement drive --dry-run`). It removes serial discovery. It also creates a green result that is easy to over-trust: three days later a green dry-run preceded FINDING #94. By design the dry-run checks preconditions only up to the first write, and the 2026-11-15 attempt failed later, at successor enrollment (Chapter 11).

A standing grant for reversible steps. It removes a signature per routine step. In exchange Stephen sees those steps in receipts after they happen, and the protection rests on the enumeration being right about what is reversible.

Runbook-scope grants. One review and one signature cover a phase. The reviewer must now hold a whole runbook in view, and a flaw in a late step is signed together with the first. The stop-on-failure rule is what bounds this.

Realistic deadlines. The intent was thirty-minute waits. The ten-minute cap on blocking waits made that impossible, so in practice the lever became: ask only when Stephen has shown a sign of life recently, notify once, and batch several decisions into one signature. Work therefore waits for his presence instead of timing out repeatedly.

A clerk-owned pause, except for adoption-class steps. Routine windows no longer need the operator's token handed over and back. The pause capability is concentrated in one agent role, and an operator-owned pause is made only immediately before a step whose code demands it.

The same evening Stephen made his signature sufficient for the mayor to run the pinned command itself, so that a signature from the phone still produces progress. Where a command attests an actor, the grant text must say that the mayor passes the name as his delegate, and the reason recorded in the database row says so too. The strongest check in the repository compares a string to `"stephen"`, and from that evening an agent may type that string when a signed text says it may. What stands between a signature and an action is then the agents' own verification of the grant, in public, before the run.

Source Notes

Claim or section	Tag	Anchor
Authority surfaces in code	[code]	`fleet_cli.py _authorized_for_all_stop`; `note_lifecycle_ops.py approve`; `tool.py _detect_source, _set_note_disposition`; `refinery_admission.py resume_admissions`; `replacement_adopt.py REQUIRED_ACTOR`; `agent_launch_environment.py`
No signature verification in the repository	[code]	Search of `yesod/src`, `docs`, `ops` at `16606a42` (research brief 11); `prime.py` "AUTHORITY INVARIANT"; `docs/security/admissions-pause-token-surfaces.md`
Same-window grant conditions	[code]	`docs/runbooks/replacement-driver.md`; `replacement_preflight.py PREFLIGHT_DOCUMENT_KIND`; commit `5299ca922`
`sjbis` as the repository sees it	[code]	`mail_cli.py` (`yesod mail ask`, `HUMAN_ADDRESS`); `prime.py HUMAN_CHANNEL_ADDENDUM`
`ta` and `sjbis` in use; the split	[notes]	Agents' working notes, 2026-10-24, 10-25, 10-26, 10-27
Grant scopes and no-grant zones	[notes]	Agents' working notes, 2026-10-10, 10-11, 10-13, 10-24, 10-29, 11-06, 11-10
The ten steps	[notes]	Agents' working notes as of 2026-11-19 (research brief 11)
Release window of 2026-11-11	[notes], [code]	Agents' working notes, 2026-11-11; commits `1c2fe38b7`, `e96ff1530` (same tree); `resume_admissions`; rule K2 in `rules_contract.py`
Signer added mid-ask; parking in silence	[notes]	Agents' working notes, 2026-10-24, 10-27
Summary signed; mayor wrote the patch	[notes]	Agents' working notes, 2026-11-08, 11-19; FINDING #58 in Chapter 17
FINDING #65	[notes], [code]	Agents' working notes, 2026-11-09, 11-11; K2 "KNOWN GAP" text in `rules_contract.py`
Four refusals; missed cue	[notes], [code]	Agents' working notes, 2026-11-10; commits `ad55cf61d`, `eef271b34`
Three taps; script versus code path	[notes], [code]	Agents' working notes, 2026-11-13, 11-18; `tool.py _set_note_disposition`; `refinery_queue.py EnqueueRefused`
Green dry-run and rehearsal	[notes]	Agents' working notes, mid-November 2026
FINDINGS #100 to #106	[notes]	Agents' working notes, 2026-11-16, 11-17
FINDINGS #127, #128, #130	[notes], [code]	Agents' working notes, 2026-11-19; `acceptance_evidence.py` (`acceptance_not_executed`, `db_env_unset`)

Claim or section	Tag	Anchor
Speed levers; delegated execution	[notes], [code]	Agents' working notes, 2026-11-10, 11-11; commits `bcedcc1cb`, `93e95c439`
What each lever trades	[inferred]	This chapter's reading of the notes

Not verified:

- How `ta` signs and verifies, what its signer registry contains, and what its time-to-live binds. The tool is outside the repository; all statements about it come from the working notes.
- The standing grant's exact text and expiry. The notes give 2026-11-24; the signed text was not read.
- The clerk's release script and window scripts. Rule K2's text on `main` names `release.sh`, but no such file was found in the repository at the snapshot; the scripts are taken to live in the clerk's role directory.
- The exact time `ys-yes-fs5w` merged to `main`. Commits `1c2fe38b7` and `e96ff1530` are ancestors of `origin/main` with identical trees, and the candidate commit's timestamp is derived from the tip's, so git does not show the time of landing. The notes place the merge between 18:42Z and about 21:45Z on 2026-11-11.
- Whether `_authorized_for_all_stop()` passes for every agent on the control machine. This is inferred from its user-name fallback and was not checked against the deployment.
- Who formally numbers findings and who rules on them. The notes appear to show one shared sequence and no written charter; this was not confirmed.
- What the verification returned for the grant signed by the newly added device on 2026-10-24. The notes record the objection and the rule.
- What has been rehearsed off production is as the working notes record it (Chapter 11).
- The three production replacement attempts of mid-November are told in general terms in Chapter 11; this chapter refers only to the last.
- All durations, counts and times of day attributed to the working notes are single-source and were not checked against the bulletin or the database.

19 What Failed, August to November

This chapter is the history of the autumn told through its failures, in order. It covers 2026-08-20 to 2026-11-19 in four periods: late August, September, October, and the November campaign to make the controller replace itself. The earlier chapters each tell the incident that changed their own mechanism. This one supplies what they cannot: the order of events, and how the fix for one period's problem set up the next period's problem.

Three things come out of reading the failures together. Most of them were of kinds the reliability contracts of Chapter 15 (The Reliability Contracts) do not name. The factory's guarantees about what reaches `main` held while its availability did not. And the record itself gets thinner after September, which changes how much weight a reader should put on the later half of the story.

19.1. The Shape of the Autumn

The commit history gives the outline before any incident is told. The counts below are from `git rev-list --count --since=2026-08-20T00:00:00Z origin/main`, with months by committer date in UTC. The candidate merges are the commits with the subject `refinery: deterministic candidate`, which the merge controller writes when a change lands (Chapter 8, The Merge Controller). One caution applies: a candidate commit carries a derived timestamp, one second after the later of its base and source commits (`_deterministic_identity_env` in `refinery_git.py`), so a merge is counted where its source commits fall, which is not always the day it was pushed.

Window	Commits	Candidate merges	Non-merge
2026-08-20 to 08-31	425	37	379
September	1,150	194	888
October	1,400	160	1,162
November 1 to 17	109	19	88
Total	3,084	410	2,517

Three readings matter for this chapter. First, September's 194 candidate merges are the high point; October has more commits and fewer merges, which is consistent with October's larger

notes and with a serial refinery lane that spent part of the month wedged (this is inferred, not measured). Second, the busiest ISO week of the whole period is October 12 to 18 with 549 commits, which is the same week as the merge-turn incidents, the second schema outage and the decision to change how deployment works. Third, November's fall from 1,400 commits to 109 has a cause that the counts alone do not show: according to the mayor's working notes, admissions to the refinery were paused for most of the month around controller replacement windows and everything else was parked. That reading is inferred from the counts plus those notes.

19.2. Late August: Contracts and Re-anchoring

The August edition of this book closed on a success. The four-host in-place promotion had completed after eleven attempts with no corrupted state (Chapter 15 keeps that campaign as dated history). The first week after it produced three small failures of the same family, all documented in commit bodies on `main`.

On 2026-08-21 a screening feed was found to have screened nothing for its whole life. Its model was pinned to a path the provider answered with 404, every call failed closed to `unavailable`, and the result looked exactly like a provider outage; the commit body says zero provider spend is what finally surfaced it. The fix added a self-test that separates misconfigured from down (`c36105967`). The same day a promotion rolled back because systemd killed a service at its 120-second start limit while the service was still doing useful work; the limit became 1,800 seconds and the test now asserts a floor (`5751aa4eb`). Both are fail-closed behavior working as designed and still costing availability, and most of the incidents below have that shape.

Two decisions in the same fortnight shaped September. On 2026-08-24, ADR 008 ended the practice of building the hosted product on a long-lived side branch: by then `main` had grown the real planning, arming and refinery machinery and the branch described a parallel control plane. From then on hosted work merged to `main` dormant, through the same serial gate as everything else (Chapter 21, The Road Ahead). And on 2026-08-28 the first commits moving the merge gate onto the home hypervisor were committed (`1964730c3`, "make Sefer the pinned primary gate runtime"). The note behind them records a candidate that passed three pristine CPython 3.14.6 gates and went `gate_red` in the cloud MicroVM gate (Chapter 9, The Deterministic Gate). More work was now headed for one gate, and the gate was about to be rebuilt.

One more August item rests only on the agents' working notes and has not been verified: on 2026-08-26/27, after a host crash, an autonomous coordinating session restarted on its own, resumed refinery admissions and enrolled a candidate. The gate run was red and nothing merged, but an operator had to restore the pause by hand. Chapter 17 (The Agent Organization) describes the singleton protections that address this class.

19.3. September: The Gate Comes Home

September's main work was the gate fleet on the hypervisor sefer: nine disposable gate VMs and their databases, built largely by hand; the gate-fleet migration epic alone accounts for 136 commits between 2026-09-13 and 09-27 (`git log --grep` for its bead id). The serial full-suite gate ran for the last time on 2026-09-01 (ADR 009). What the factory gained was a gate it controlled. What it took on was every property of one physical host: name resolution, credentials on temporary filesystems, database service units, memory. The first full day of living with that was 2026-09-09.

> **Factory Floor — Eight silent failures in one day (2026-09-09).** The agents' working notes for this date list eight separate faults, every one of which presented as healthy: a held lane slot, unresolvable gate hosts, lost credentials, databases down behind a unit that said `active`, and four measurements that were silently empty or unenforced. Chapter 13 (Seeing the Whole System) gives the list and checks it against source. The lesson the notes draw is that the absence of a bad signal was being read as a good one. A heartbeat proves a thread is alive and says nothing about whether the work loop behind it is doing anything.

Two other events of the same day belong with it. Pointing the topology at the new gate fleet took the control plane down, because two copies of a hostname pattern refused the new names at startup (`2d7e45332` replaced them with one shared pattern). And the notes record a near-miss of the "unknown read as zero" kind: a check for unharvested work on a seat was run as the wrong user, every git command failed silently, and the count came back as zero uncommitted and zero unpushed. That zero would have authorized destroying the seat. Rerun as the owning user, the same check found an unpushed commit and 290 lines of documentation.

The next day's incident is told in Chapter 6 (The Runner): the runner's dispatch watchdog kept restarting runners that were holding work, and the drain timeout then killed in-flight agents (fix `6330a83aa`). One detail belongs here. The agent diagnosing it first blamed memory recycling, from a code comment next to the shutdown path, and found the real trigger hours later in a log table it had not queried.

A second kind of self-inflicted damage came from recovery.

> **Factory Floor — Stranded repairs that were operator damage (2026-09-09 to 09-11).** Four conflict repairs looked stuck, two of them for about 40 hours. The status history told a different story. In each case arming had worked: the note moved from `awaiting_merge` to `open` within 8 to 12 seconds, which is the state in which a runner can claim an armed repair. Then a recovery procedure, run by an agent acting as operator, moved each note to `captured` and re-planned it, ending at `planned`, where an armed repair can never be claimed. For two notes that recovery came within five minutes of arming; for the other two, 5.5 hours later. The recovery had also reset each note's automation policy from `deliver` to `plan`. Separately, two of the repairs had been armed to a dispatch lane (a model target) that could not run at the time. Once the policy and status were corrected on 2026-09-11, all four ran within about 40 seconds. The code change of the period is `7c8ff9b41` (2026-09-11), which made the repair row itself the claim authority (Chapter 8); armed repairs are claimed through `claimable.repair_maintenance_candidates`. The notes say plainly that it would not have freed these four on its own. All timings here are from the agents' working notes. The lesson: before recovering something, check whether it is already in the state where the machine is about to act on it.

The same investigation found a state nothing detected. Four notes had every child bead closed, a branch ahead of `main`, and no path to merge: lifecycle `planned`, a terminal bead graph, nothing to dispatch and nothing to move them forward. The notes call this "finished but unbanked" and give the branches as between 85 and 736 commits behind `main`. One consequence was hours spent arguing whether to strike a dependency edge between two notes, when the upstream note was finished and merely unmerged. The mechanisms came afterward: `completed_plan_reconciliation.py` reconciles idle completed plans from durable evidence (`1e82ec892`, 2026-09-15), and a detector for stranded completed plans followed on 2026-10-04 (`207f3b3ec`). The related trap, a note that displays `planning` while its planner job is dead because a helpful comment superseded the plan, is told in Chapter 5 (Capture, Plan, Deliver).

The gate's own incidents of the month are in Chapter 9: an operator merged a change past a gate judged falsely red on 2026-09-11 (`8326d9a3c`), and three report readers were found to count red runs as green (`cfb66cf03`, committed 2026-09-20 UTC). Chapter 1 (Follow the Change) tells how two pull requests merged outside the gate left `main` red on 2026-09-14 (`0373aa5b4`).

Meanwhile the deployment side changed in a way that matters for October. On 2026-09-16 a fresh VM, VM175, replaced the original controller VM by hand, and on 2026-09-17 a whole fleet release was applied by hand in one pause window. The runbook for that release ends by saying that no routine merge-to-fleet transport existed yet and merges would strand until an operator repeated the procedure. Runbooks on `main` dated 2026-09-18 to 09-20 then record the activation of a routine path for runner seats only (they are plans and sentinels, and do not record the first completed apply): seats take each merged release and the controller does not. The in-place failures of that same week are told in Chapter 10 (Merged Is Not Deployed). On 2026-09-20 the release surfaces sat at 0, 35 and 529 commits behind `main` at the same time, which produced `yesod release-skew` (`7bb582e12`, whose commit body gives the figures). The asymmetry between seats that roll forward on every merge and a controller that waits for a hand window is the direct cause of October's two schema outages.

September ended with the host itself.

Factory Floor — The host hard-reset (2026-09-27 17:52Z). The hypervisor sefer reset with no kernel panic and no out-of-memory record. According to the agents' working notes, configured guest memory exceeded physical memory (262 GB against 251 GB), the host had no swap, and the start of a merge gate run filled the eight gate VMs to their full allocation. Userspace stalled, a high-availability software watchdog missed its deadline and reset the machine. The host was back in about five minutes, and the factory was not: only shared services and the controller start on boot, so runner seats were started by hand at 20:28Z and the gate fleet stayed stopped. Every VM came back with new SSH host keys, which made every pinned key stale (Chapter 12, Born, Not Repaired, tells the cause and the fix). The changes recorded in the notes: compressed swap added and the watchdog disarmed the same evening, and host memory raised to 376 GB in an early-October window. On `main`: the runbook `sefer-ssh-host-key-refresh.md` (`211262083`) and an advisory headroom policy (`558927db3`). The 48-hour soak of the new gate fleet had closed that morning with 1 of 3 required green merges and storage at 79 percent against an 80 percent limit (`docs/runbooks/soak-9114-2026-09-25.md`). The gate had come home to a host with no margin.

19.4. October: Merged, Not Deployed

October opened with the memory fixed and the bottleneck moved. The mayor's notes for 2026-10-10 name the serialized gate, at about 70 minutes per candidate, as the throughput ceiling, with runners idle behind it. Under that pressure the gap between merged code and running code became the month's subject. The first outage came from the routine deployment path September had built.

Factory Floor — About nineteen hours fleet-down (2026-10-07/08). A routine rollout staged runner code that needed a migration the database did not yet have (Chapter 10 tells this cause and its recurrence on October 13). Every dispatch loop wedged about eight minutes after start, from roughly 07Z on October 7 to 03:55Z on October 8. Another agent sent an urgent mail at 07:15Z; the mayor's notes say it went unread for hours. Recovery was a hand promotion of the controller, which applied the migration, and a restart of the runner service on six seats, each taking about five minutes because a wedged loop ignores the stop signal until its timeout. Service did not return with the runners. Four seat gateway keys had expired during the outage, so planner jobs failed with no result until the keys were extended. Then plans failed with `completed_prefix_invalid`, because a run that closes several children stored only the last child's commit; `e9bb119d4` fixed the validator. Then the next merge failed to deploy because new dependencies were missing from an offline package cache, and three merges ended `merged_not_deployed` behind it. The duration and sequence are from the agents' working notes; only the validator fix has a commit on `main`. The lesson is that an outage of this length is seldom one fault. Each recovery step exposed the next thing that had expired or drifted while the fleet was down.

The following week brought four more, each told in the chapter that owns its mechanism. On 2026-10-12 a `gate_green` attempt twice failed to push because a later claim took its refinery lane's merge turn (fix `779673ec0`; Chapter 8). On 2026-10-13 the runners rolled ahead of the schema a second time. On 2026-10-14 a promotion replayed an old constraint against

rows that a later migration had made legal, minutes after a clean full pre-run of the same payload; the class fix made migration replay missing-only (`9937e15f1`). On 2026-10-15 a nightly CI job was found to have migrated production outside any window (`34e1d4085` and `968bbb34a`, then `c3451be82` on 2026-10-30 making migration opt-in everywhere). Chapter 10 tells the last three. The commits carry subjects only, so what happened on each date is from the agents' working notes.

The state `merged_not_deployed` had been added on 2026-10-03/04, and it made the scale visible. The agents' working notes record 15 of 15 merges ending in that state in the 24 hours around 2026-10-13, and 51 parked by 2026-10-14, when a controller guard that assumed "succeeded" meant "latest pushed" deadlocked the lane (Chapter 8). On 2026-10-13 Stephen set the rule that ends Chapter 10: all deployment should be stopping one machine and starting a different one, with a short handover. The replacement framework was committed within two days (`8063fbcc1` and its siblings) and, per the notes for 2026-10-14, had no production caller yet. The driver, adoption and successor protocol followed between 2026-10-26 and 10-30 (Chapter 11, Deploy Is a Host Swap).

The other October change was organizational. To protect the serial refinery lane, every delivery was held, given a full test pass on a disposable machine, and only then released. That procedure was a set of agent sessions, and it failed in its own way.

On 2026-10-27/28, according to the agents' working notes, the "hold, full pre-run, release" step stopped happening. The agent session that owned the pre-run was not running, releases went straight into the refinery lane, and the lane's gate became the first full test run each change received. The notes estimate about seven lane-hours of red gates, mostly test and fixture misses, before Stephen asked why merging was slow and restarted the seat. The hold reason on the notes still read as awaiting a pre-run while releases went ahead. No commit answers this incident because nothing in code owned the step. A process step that lives in an agent session has no alarm for the session ending.

On 2026-10-29, the notes record, a builder agent wrote test fixture rows into the production replacement store (FINDING #58), which produced the rule that agents on seats next to production do not author application code (Chapter 17).

19.5. November: The Swap Campaign

By the end of October the automatic swap existed on `main` and had never run. November was the attempt to run it once, and the method chosen was conservative: admissions paused, each step executed inside a window under a signed single-use grant, with independent review before each signature. The factory's serial lane had been the bottleneck in October. In November the bottleneck was a serial sequence of human approvals on paths that had never executed.

The month began with a hand-run cutover to a new controller VM over three sittings, serving from 2026-11-03 00:59Z (notes). It then lost most of a day to a fault with nothing to do with the swap.

On 2026-11-08 the planning queue looked frozen: notes sat in `planning` and queue counts did not change between checks. The planner and build seat, gilgul, had been stopped for 15 hours

45 minutes after a power event, and VMs of that kind do not start on boot by convention. The mayor's notes say it spent about two hours reasoning from job-state snapshots and a gateway dashboard, and that the devil's advocate followed the same wrong signal for about ten hours. The forensics agent had flagged the seat as dead that morning, when nothing yet depended on it, and nobody raised it again when work did. Stephen started the VM. A request for an alert on "queued work and no live runner able to take it" was filed; no code change on `main` corresponds to it. The supported way to inspect planner jobs was documented the next day (`7c8d262da`, `docs/runbooks/planning-job-inspection.md`). The account and all its durations are from the notes.

The swap work itself is told in Chapter 11, and its governance in Chapter 18 (Signed Human Authority). In outline, from the agents' working notes: on 2026-11-10 adopting the serving controller into the replacement lineage was refused four times, each on a precondition readable in the code beforehand, before succeeding at 23:20Z. The response was a read-only preflight that reports every failed precondition at once (`bcedcc1cb`), green in production on 2026-11-12. In mid-November 2026 three replacement attempts were submitted by hand in production. Each stopped during the successor's birth or enrollment, before any transfer of authority, on a different precondition that only a real run exposed. Each ended `needs_operator`, the serving controller kept serving throughout, and each was followed by a root cause and a fix before the next request. The last of them followed a day of reviews and a green dry run, and a rehearsal on a scratch machine afterwards found further blockers.

The fix for those blockers is one note, `ys-yes-gn4b`, and its path through the next three days shows every failure class of the month in sequence. It was filed as prose and built on the isolated seat: `captured`, `planning`, `planned`, `open`, `in_progress`, then `awaiting_merge` with disposition `held`. On 2026-11-16 a window opened and its attempt ran `requested`, `claimed`, `preparing`, `gating`, `gate_red`: one test had passed in pre-run and failed at the gate because a different test leaked a process umask. Rework followed, during which three defects passed four reviewing agents and were caught only when code was run, and a wrong cause named from a text search cost a full rework cycle (Chapter 18).

Factory Floor — Three merge windows stopped in one night (dated 2026-11-18 in the notes). Three consecutive windows for the same note stopped before any attempt existed. The first stopped on a hold sentinel the window's text had not listed. The second stopped on a stale queue row (FINDING #107). The third stopped at `yesod refinery enqueue --strict`, which refuses with `EnqueueRefused` (reason `admission_paused`) while admissions are paused: the reordered steps reached the enqueue before the resume, and the agent preparing the window had written that the release wrapper script had no dependency on admissions after reading the script and not the verb it calls. `require_admissions_open` guards enqueue in `refinery_queue.py`; the devil's advocate had named the risk and it was not checked. Nothing merged and nothing was damaged, and each stop spent a single-use grant and required a new signature. The fourth window worked, with one unplanned step: the refinery's own reconcile pass enrolled the `awaiting_merge` note 1.8 seconds after admissions resumed, before the script did. The attempt ran `gating`, `gate_green`, `pushing` and ended `merged_not_deployed`, because the controller still runs the older release. That merge is the candidate commit `16606a42`, the tip of `main`. It is stamped 2026-11-17 19:16Z, a time derived from its source commit `5451304da` and not the time of landing; the working notes place the merge window on 2026-11-18. Sequence and timings are from the agents' working notes. The lesson recorded: for every step whose order changes, trace the called verb into the library, and trace every other writer of the state the step creates.

The last finding of the period came on 2026-11-19. All nineteen children of the next hardening note had closed, and every close reason read `acceptance_not_executed` with reason `db_env_unset`: the build environment has no database, so database-backed acceptance never ran, and the children closed anyway (FINDINGS #127 and #128; Chapter 7, Governed Agent Execution). The strings are in `acceptance_evidence.py`; that every child carried them is from the notes.

19.6. What the Failures Have in Common

The five rules of Chapter 15 are about evidence for a claim: name it, refuse without it, journal around effects, re-prove on recovery, bind to exact identity. Nothing found on `main` contradicts them. Most of what stopped the factory this autumn was of four other kinds.

Alive but doing nothing. A lane slot held by a finished attempt, a database behind a service unit that says `active`, a stopped VM behind a queue that says `planning`, a process step whose agent session has ended, a completed plan with no path to `main`. In none of these was any evidence wrong. Evidence of progress was missing and nothing was required to notice. The fixes that exist are bounds and detectors (`deployment_settle_timeout`, the stranded-plan detector); the wall-clock limit is enforced only by the runner that owns the agent process, and the missing-runner alert is still open.

Skew between surfaces. Code, schema, the controller's configuration document and the installed release moved at different times. Both fleet outages in October, the configuration wedge and the nightly migration are this kind. `release_skew.py` measures it, `merged_not_deployed` names it, and `db.init_db` now refuses to migrate without `YESOD_ALLOW_MIGRATE`. Closing it is what the swap is for.

Replay hazards. A heal or retry re-executed something that was correct when first run and wrong now: an old CHECK constraint replayed over newer rows, a baseline replay that contended with a backup, recovered work killed again by the same watchdog. Missing-only migration replay is the one class fix.

Coordination failures between agents and the human. Wrong causes stated with confidence, recoveries that damaged a working state, grants spent on commands nobody had executed, reviews that read a wrapper and not the code under it. This class has no states in the database and the weakest mechanism. It accounts for a large share of the lost time in this chapter.

On integrity and availability the record splits. No record was found of a bad change reaching `main` through the gate, or of corrupted lifecycle rows; this is inferred from the chronicle, not proven by an audit. The exceptions are near the edge of that claim and should be read as such: `main` went red through a door that bypassed the gate, one merge was deliberately landed past a gate judged falsely red, fixture rows reached a production table through a builder's interpreter, and a CI job migrated production. Availability did not hold. By the working notes' figures, the factory was stopped or wedged for about nineteen hours on October 7 and 8, held repairs frozen for about forty hours in September, and waited most of a day on a stopped VM in November. `main` carries 19 candidate merges dated in the first seventeen days of November. Fail-closed design kept the stops safe and did not make them short.

The evidence for all of this is uneven, and a reader should know how. Non-merge commit bodies on `main` total 238,589 bytes for September and 37,250 bytes for everything from October 1 onward (`git log --no-merges --format=%b | wc -c`, by committer date in UTC). In September the commits were largely hand-written and their bodies cite the live incident, with numbers. From October most commits carry a subject line only (1,183 of the 1,250 non-merge commits have an empty body), and the reasons moved to notes, the bulletin board and the coordinating agents' working notes. A search of commit messages since 2026-08-20 finds the word "postmortem" zero times. So the September half of this chapter can be checked against `main`. The October and November half rests mostly on what agents recorded at the time, corroborated by commit subjects and dates. Those notes are candid and they are written by the parties whose mistakes they describe; they include their own retractions, as with the watchdog. Every number in this chapter that comes only from them is marked, and none should be quoted as measured.

19.7. The Evening of 2026-11-19

The last entry in this history was written the day the edition was assembled, and it comes from the mayor's board notes. The hardening work for the fourth attempt, H1, passed its full Linux test union that day (4,591 passed, 0 failed, on a disposable test container). In the same evening, before any production step, pre-flight and rehearsal raised seven findings, #130 to #136. Chapter 11 lists them. Three are worth carrying into this chapter's argument.

The fourth submit would have been refused at its first command (#130): the proof that clears the three earlier failed intent generations compared their recorded migrations with the new candidate's list, and H1 adds a migration. A read-only run against production's records found it. Every test box had used one migration ladder.

No successor could have finished enrolling (#136): H1's enrollment check required three services to be active that the birth fence deliberately holds down until activation. It passed review and all 4,591 tests because the test stand-in answered "active" for every unit. The first real birth on a rehearsal box found it.

And the goal's final step deadlocks as coded (#132): a merge-triggered swap waits on a quiet-gap proof that counts its own triggering attempt as in flight. Every swap in production and in rehearsal had been submitted by hand, so nothing had run the path. The reading came first from the fact-check of Chapter 11 of this book and was then confirmed by two reviewers.

These are the four kinds of failure again, in one evening. A test double that was alive and proved nothing. Skew between one migration ladder on test boxes and several in production's records. A path nobody had run. And two approval requests for an agent-written patch made on a partial test run, both replaced, after which the rule became "no approval request until the full Linux union is green on the exact tree". None of it touched production.

19.8. Status on 2026-11-19

As of the evening of 2026-11-19 the first fully automatic, merge-triggered controller swap has not happened in production, and it has never run anywhere, including in rehearsal. No replacement intent has reached `succeeded` in production, and the handover has never run there. The three production attempts of mid-November were submitted by hand; all ended `needs_operator`, none transferred authority, and VM179, serving since the hand cutover of 2026-11-03, kept serving throughout (notes).

Chapter 11 gives the full account of what is proven where. In short: adoption (2026-11-10 at 23:20Z) and the read-only preflight (green on 2026-11-12) have run in production, per the agents' working notes. The first-stage fixes from the rehearsal are on `main` at `16606a42`, merged and not deployed. The hardening patch is on its fifth revision, still in its test gate and not on `main`; it carries a migration, which must go through a hand window because automatic swaps refuse migration-bearing candidates (`AutomaticMigrationRefused`). No fourth attempt has been made. Hand-submitted swaps have been rehearsed on disposable machines using reference code. The first birth of the built hardening code on a rehearsal box was on 2026-11-19 and is what exposed finding #136. A merge-triggered swap has never run anywhere.

Source Notes

Claim or section	Tag	Anchor
Commit table and weekly counts	[code]	`git rev-list --count --since=2026-08-20T00:00:00Z origin/main`, months by committer date in UTC; `_deterministic_identity_env`

Claim or section	Tag	Anchor
November pause as cause of the fall	[inferred]	Counts plus agents' working notes, 2026-11-10
Late August feed, start timeout	[code]	`c36105967`, `5751aa4eb` (commit bodies)
ADR 008; gate comes home	[code]	`decisions/008-...re-anchor.md`; ADR 009; `1964730c3`; `docs/tools/yesod.md`; Chapter 9
Restarted coordinating session	[notes]	Agents' working notes, 2026-08-26/27
Eight silent failures	[notes]	Agents' working notes, 2026-09-09; Chapter 13
Hostname pattern outage	[code]	`2d7e45332` (commit body)
Empty read as zero	[notes]	Agents' working notes, 2026-09-09
Watchdog	[notes], [code]	Notes 2026-09-10; `6330a83aa`; Chapter 6
Stranded repairs	[notes], [code]	Notes 2026-09-11; `7c8ff9b41`; `claimable.py`
Finished but unbanked	[notes], [code]	Notes 2026-09-11; `1e82ec892`, `207f3b3ec`
Hand controller and fleet release; seat activation	[deploy]	Runbooks dated 2026-09-16 to 2026-09-20 on `main`; Chapter 10
Three surfaces, three releases	[code]	`7bb582e12` (commit body); `release_skew.py`
Host hard-reset	[notes], [deploy]	Notes 2026-09-27, 2026-10-03; `211262083`, `558927db3`; soak runbook
Gate as throughput ceiling	[notes]	Mayor's notes, 2026-10-10
Nineteen hours fleet-down	[notes], [code]	Notes 2026-10-08; `e9bb119d4`
October skew and replay incidents	[notes], [code]	`779673ec0`, `9937e15f1`, `34e1d4085`, `968bbb34a`, `c3451be82`; Chapter 10
15 of 15 and 51 parked merges	[notes]	Agents' working notes, 2026-10-13 and 10-14
Stephen's 2026-10-13 rule	[notes]	Agents' working notes, 2026-10-13 (paraphrased)
Replacement framework dates	[code]	`8063fbcc1`, `8001389bc`; `git log`
Pre-run step vanished; FINDING #58	[notes]	Agents' working notes, 2026-10-28, 10-29; Chapter 17
Frozen queue was a stopped VM	[notes], [code]	Notes 2026-11-08; `7c8d262da`
Adoption, preflight, attempts	[notes], [code]	Notes, 2026-11-10 to mid-November; `bcedcc1cb`; Chapter 11

Claim or section	Tag	Anchor
The `ys-yes-gn4b` trace	[notes], [code]	Notes 2026-11-15 to 11-18; `AttemptState` in `refinery_state.py`; `5451304da`
Three windows in one night	[notes], [code]	Notes 2026-11-18; `require_admissions_open`, `EnqueueRefused` in `refinery_queue.py`; `refinery_git.py`
Children closed without tests	[notes], [code]	Notes 2026-11-19; `acceptance_evidence.py`
Four kinds of failure	[inferred], [code]	Brief 10 sections 1 and 6; `_wait_with_governance` in `runner.py`
Integrity versus availability	[inferred]	Chronicle; `8326d9a3c`, `0373aa5b4`
Commit-body volume; “postmortem”	[code]	`git log --no-merges --format=%b | wc -c`; `git log -i --grep`
Status	[deploy], [notes]	OUTLINE status facts; `AutomaticMigrationRefused`

Not verified:

- Every figure attributed to the agents’ working notes: the counts and durations in the eight silent failures, the stranded-repair timings, the branch distances of the unbanked notes, the memory figures and times of the host reset, the nineteen-hour duration and its cascade, the seven lane-hours, the 15 hours 45 minutes, the 1.8 seconds.
- The 2026-08-26/27 restarted coordinating session. Two briefs describe it differently; no source on `main` was found.
- The date of the host memory upgrade: one set of notes gives 2026-10-08, another 2026-10-09.
- The time the fourth window’s merge was pushed. The working notes place the merge window on 2026-11-18; the stamp on the candidate commit `16606a42` is derived from its source commit and is not a push time.
- Why October has more commits and fewer candidate merges than September; the reading given is an inference.
- What exactly the “unlisted sentinel” of the first stopped window was; the notes give only that phrase.
- The evening of 2026-11-19 (findings #130 to #136, the test count, the state of the patch) is as reported in the mayor’s board notes; only the quiet-gap blocking states and the birth fence were checked on `main` (Chapter 11).
- Whether the claim that the runner status command hides dead runner rows (a contributing factor on 2026-11-08 per the notes) is true on `main`; it was left out of the text.
- Whether the off-site handover rehearsal has completed, and the current content and state of H1.
- That no bad merge reached `main` through the gate is a reading of the chronicle, not the result of an audit.
- The numbered findings live on the bulletin board, which was not read; finding numbers are as cited in the working notes.

Part VI.

Beyond the Current Factory

20 Patterns Worth Reusing

Yesod is one implementation. This chapter lists the ideas in it that can be adopted without copying its roles, schema or host names. The first fourteen patterns come from the earlier editions and still hold; a few have been corrected where the mechanism changed. The eleven after them were earned between August and November 2026, mostly by failing first. Each of those carries one Yesod example and the chapter that tells the full story.

Two cautions apply to the whole list. Some of these patterns are enforced by code in Yesod and some are held only by discipline, and the text says which. One of them, replacing hosts, has not yet completed automatically in production for Yesod's own controller.

20.1. Durable Intent Before Automation

Start with an addressable work record that preserves the request, the acceptance evidence, dependencies, the routing decision, every attempt and the final disposition.

Do not begin with a swarm. A system that cannot explain why a task exists becomes less understandable when ten agents work on it at once.

20.2. Separate Work Graph from Execution History

A work graph answers what depends on what. An execution ledger answers who tried, when, with which model, and what happened. They change at different rates. Folding them into one mutable status field makes retries, partial progress and parallel work hard to represent.

20.3. Make Dispatch Orthogonal

Where a note is in its life, whether something is holding it, and which model target may run it are three facts. Yesod stores them as a status, a disposition (`active`, `held`, `blocked` and others, each with a reason) and a dispatch arm written by a scheduler after approval.

Keeping them apart allows explicit holds, re-routing without rewriting the lifecycle, and an answer to "why is this approved note not running" (Chapter 5, Capture, Plan, Deliver).

20.4. Claim Atomically, Repair Cross-Store Effects

Use a short transaction and a row lock for the authoritative claim. Perform slower external effects after the lock is released, and compensate visibly if one fails. Two independent stores do not share a transaction, and explicit repair is safer than pretending they do.

Choose the weakest lock that is correct. On 2026-09-25 a claim that locked its runner row `FOR UPDATE` waited on its own log insert from a second connection; the fix was `FOR NO KEY UPDATE` (`10090d261`; Chapter 6, The Runner).

20.5. Give Every Run a Disposable Workspace

A Git worktree gives isolation without a virtual machine per attempt. It needs a known base revision, a unique branch, no shared uncommitted state, a durable path for forensic recovery, and clear cleanup rules. The workspace is disposable. The commits and logs are not.

20.6. Put Governance Outside the Model

Wall-clock limits, token budgets, kill requests, process-tree termination and remote-branch proof belong in a governing service. In Yesod the runner's `_wait_with_governance` loop enforces the first four on the agent process it launched (the token budget only when one is configured), the runner pushes and verifies the remote branch itself after the agent exits (`_push_and_verify_branch`), and `build_agent_launch_environment` gives a dispatched agent only an allowlisted environment. The worker prompt explains the rules; code enforces them.

Record the limits that cannot be enforced as such. Yesod marks a configured spend cap `not-enforceable` on the run because no backend reports cost while the run is live (Chapter 7, Governed Agent Execution).

20.7. Define Proof of Work

Choose proof that fits the task: commits past a known base for code, a passing acceptance probe for data, a fork branch and pull request for a remote contribution, a receipt naming the exact candidate for a deployment. Exit codes and transcript language prove nothing.

Proof also has a scope. A closed child task proves a commit exists. It does not prove a test ran (see "Run the Tests in the Audit" below).

20.8. Use One Automated Integration Door

Workers propose branches. A separate deterministic service builds the exact candidate against current `main`, runs the gate on that candidate, lands it with a compare-and-swap push, and records what landed. That keeps the credentials and contexts that can mutate shared history to one narrow place.

The door protects `main` only while it is the only door. On 2026-09-14 two pull requests merged outside the gate left `main` unable to pass it, and every merge stopped until the repair `0373aa5b4` (Chapter 1, Follow the Change).

20.9. Distinguish Red from Unavailable

A test failure is information about the candidate. A gate that could not run is information about the infrastructure. They need different counters, retry policies and operator responses. Treating unavailable as red punishes good code; treating red as unavailable retries forever.

The same rule applies to any reading: an unknown is never a zero. A counter that cannot see a class of result must not back a verdict (Chapter 9, The Deterministic Gate; Chapter 15, The Reliability Contracts).

20.10. Reconcile Durable Scratch

Any queue that survives process death needs a repair loop for duplicates, missing sources, work that is already complete, reverted work, stuck intermediate states, and a blocked head that starves the rows behind it. A queue that persists also keeps stale rows, and reconciliation is the cost of that durability.

20.11. Record Reasons, Not Only Decisions

A lane assignment without a rationale cannot improve a policy. A hold without a reason becomes debris. A silent skip is worse than either: on 2026-09-17 a second project sat queued at attempt zero because an “unavailable” exception was swallowed with no reason (`2d4759879`). Store the why next to the what, including for the things the system declined to do.

20.12. Price Delivered Outcomes

Capture tokens and cost per attempt, keep unknown values unknown, and roll every attempt into the delivered change. The quantity worth comparing is cost per accepted change.

Yesod captures this and does not yet act on it. No code on `main` reads cost or complexity to choose a model (Chapter 16, Cost-Aware Dispatch).

20.13. Make Authority a Data-Flow Property

Do not infer authority from a display name or a message sender. Define the channel through which human authority enters and refuse to upgrade peer coordination into commands. Then be honest about which parts of that channel are code. In Yesod few are (Chapter 18, Signed Human Authority).

20.14. Date the Fleet

Source-grounded documentation and deployment documentation are different products. Put a commit on the first and an observation time on the second. On 2026-09-20 the operator command line, the controller and the build seats were 0, 35 and 529 commits behind `main` at once, and each answered confidently; `yesod release-skew` now measures the distance (`7bb582e12`; Chapter 13, Seeing the Whole System).

20.15. Replace Hosts Instead of Mutating Them

A host that is changed in place drifts from the recipe that built it, and each in-place deployment is a new procedure run against unknown state. A host that is born from a template, proven, and swapped in exercises the same recipe every time.

Yesod reached this by failure. The agents' working notes for 2026-10-13 record 15 of 15 merges in 24 hours ending undeployed on settle timeouts under in-place promotion. Gate machines and runner seats are now born from versioned templates with receipts (Chapter 12, Born, Not Repaired).

The controller swap is designed and on `main` and has not completed in production as of 2026-11-19 (Chapter 11, Deploy Is a Host Swap). Adopt the pattern knowing that the last host to become replaceable is the one doing the replacing.

20.16. Make "Merged but Not Running" a State

If the system can merge code that is not yet running, give that condition a name and a row. Otherwise it appears only as confusion.

The agents' working notes for 2026-09-10 and 09-11 record a merged fix to a plan validator that changed nothing, because runners execute a pinned release, and the failure message was identical before and after the merge. Yesod now ends such an attempt in the terminal state `merged_not_deployed`, projects the note to `blocked`, and opens an operator item (Chapter 10, Merged Is Not Deployed).

The state has a cost. Notes that depend on a merged, undeployed prerequisite do not unblock on their own.

20.17. Own Your Pause and Prove Quiet

A procedure that needs the system quiet should take the stop itself, record that it owns it, and prove that nothing is in flight. It should never borrow a pause someone else took for another reason.

`acquire_replacement_stop_guard` does all three in one transaction. When a foreign pause exists it takes nothing and returns a proof that is not ready, with the reason `admissions_paused_by_other`, and the handover raises `ControllerHandoverDeferred`. Resuming needs the pause token and the expected revision (`resume_admissions`).

"Paused" must also say what it fences. Yesod's admissions pause is checked at intake and enqueue only (`require_admissions_open`). According to the agents' working notes, on 2026-11-09 a migration note nearly merged outside any approval because an attempt was created before a hold applied (Chapter 8, The Merge Controller, and Chapter 18).

20.18. Put the Point of No Return in Writing

Every multi-step change has a step after which undo is no longer safe. Name it in the code and in the runbook, make every step before it automatically reversible, and make automation refuse to roll back after it.

In the controller swap that step is the `release_service_addresses` effect. `rollback` raises once that effect exists in any evidence state, admissions stay paused, and a human decides (Chapter 11). As of 2026-11-19 this boundary has never been crossed in production.

20.19. A Dry Run That Reports Every Failed Precondition

A check that stops at the first failure turns five missing preconditions into five attempts. When each attempt needs a human approval, that is five approvals.

The agents' working notes record that adopting the serving controller was refused four times before it succeeded on 2026-11-10, each time on a precondition readable in the code beforehand. `yesod replacement drive --dry-run` now collects every failure into one read-only document, `yesod.replacement_preflight.v1` (`bcedcc1cb`, `93e95c439`; Chapter 11).

A dry run checks preconditions. It does not execute the path, which is the next pattern.

20.20. Rehearse a Never-Run Path

Review, a pre-mortem and a green dry run do not substitute for running a path end to end once on a machine that can be thrown away.

In mid-November 2026 a production replacement attempt stopped after a day of reviews and a green dry run. According to the agents' working notes, a rehearsal on a scratch machine

then found further blockers. It had been declined earlier to save time (Chapter 11; Chapter 19, What Failed, August to November).

Rehearsals also expose fakes. According to the working notes, one found that test doubles returned a timer property that real timers do not have.

20.21. Separate Attention from Permission

Asking a human to look and asking a human to authorize are different requests, and their timeouts must mean different things.

Yesod's agents use one external tool, `sjbis`, for attention: a question may carry a stated recommendation and say that the recommendation stands if no answer comes within a set time. They use another, `ta`, for permission, where a timeout carries no authority at all.

Neither wait may end in silence. According to the working notes, on 2026-10-27 two requests timed out overnight and the coordinating agent parked; Stephen assumed it was working (Chapter 18).

20.22. Sign Exact Text

An approval covers the text that was signed and nothing else. The request should therefore be the full scope: the command, every script pinned by hash, the commit, the executor, the stop conditions, and whether it may be used once.

According to the working notes, on 2026-11-08 a request put a short summary in the signed field and the pinned scope in a detail field. Verification against the scope failed and the work waited for a new request. The rule since: a plain summary first, then the pinned text, all of it signed, verified before anyone cites it, and never reused for a retry (Chapter 18).

In Yesod this is discipline. Nothing in the factory repository verifies a signature.

20.23. A Rule Registry That States Its Own Assurance

Operating rules written only in prose all look equally binding. A registry that labels each rule as enforced, detected or merely attested shows where the organization is relying on good behavior.

`yesod rules check` and `yesod rules show` expose twelve rules, K1 to K12, each with an owner and an `Assurance`. Six have detectors, six are attested only, and none is `ENFORCED` (`rules_contract.py`). One rule's text records its own known gap (Chapter 17, The Agent Organization).

20.24. Run the Tests in the Audit

A review that reads a diff reports what the reviewer believes. An audit should execute the new and affected tests on the delivered commit, against a throwaway database, and name where each number was measured.

The working notes for 2026-11-16 to 11-19 record three defects that passed four reviewing agents and were caught only when code ran (Chapter 18). They also record that on 2026-11-19 all nineteen children of a hardening note had closed with `acceptance_not_executed` and reason `db_env_unset`: the build environment has no test database, so the tests never ran (Chapters 7 and 15).

The audit must never use a shared database. The notes for 2026-10-14 record a few tests run from a control machine turning a real gate run red (Chapter 17).

20.25. Dormant by Default on Main

Build a large new capability on the main branch, behind a switch that is off, instead of on a long-lived side branch. A side branch describes a system that the main branch has already left behind.

Yesod's hosted work began on a side branch and was moved by a recorded decision on 2026-08-24, because the contracts written against the branch proposed a second set of work controllers beside the ones production already had. The hosted API on `main` refuses to start unless its configuration sets `enabled` to true (`hosted_server.py`), and the replacement driver's units ship installed and disabled.

The cost is that code which has never been switched on has never met its environment (Chapter 21, The Road Ahead).

20.26. An Adoption Ladder

Teams can adopt these patterns in stages.

1. Catalog: register tools, durable notes and repeatable processes.
2. Plan: add work graphs and dependency readiness.
3. Isolate: give workers disposable workspaces and bounded governance.
4. Prove: require commits, probes and remote branch evidence.
5. Integrate: introduce a single gated merge path.
6. Observe: persist runs, gate attempts, merges and deploy outcomes.
7. Deliver: record whether merged code is running, and measure the distance when it is not.
8. Recover: add reconciliation and watchdogs that never restart a process holding work.
9. Replace: build hosts from versioned recipes, rehearse the swap, then automate it.
10. Authorize: give human permission one channel, exact text and a receipt.
11. Economize: measure cost per delivered change, then route on it with reasons.

Each stage is useful without the next. As of 2026-11-19 Yesod itself stands partway up step 9 and has not begun the second half of step 11.

Source Notes

Claim or section	Tag	Anchor
Patterns from earlier editions	[code]	`runner.py` (`_wait_with_governance`, `_spend_enforcement`); `run_ledger.py` (`SPEND_NOT_ENFORCEABLE`); `agent_launch_environment.py`; commits `10090d261`, `0373aa5b4`, `2d4759879`, `7bb582e12`
Replace hosts; merged but not running	[code], [notes]	`refinery_controller.py`, `refinery_intake.py`; agents' working notes, 2026-09-10, 09-11 and 2026-10-13
Own your pause; point of no return; dry run	[code]	`replacement_guard.py`, `refinery_admission.py`, `replacement_handover.py`, `replacement_cli.py`, `docs/runbooks/replacement-driver.md`
Rehearsal; attention and permission; exact text; audits	[notes]	agents' working notes, 2026-10-14, 10-27, 11-08, 11-09, 11-10, 11-15 to 11-19
Rule registry	[code]	`rules_contract.py` (`RULES`, `Assurance`; six `DETECTED`, six `ATTESTED`, recounted at `16606a42`), `rules_cli.py`
Dormant by default	[code]	`decisions/008-aws-hosted-production-factory-re-anchor.md`; `hosted_server.py`; `hosted_config.py`

Not verified:

- The counts and durations taken from the agents' working notes (for example 15 of 15 merges) were not measured from source.
- Dates and sequences of the incidents taken from the agents' working notes (2026-09-10/11, 10-14, 10-27, 11-08 to 11-19) were not checked against any source on `main`.

21 The Road Ahead

The August edition ended with a chapter of speculation about a learning ecology of tools. This chapter replaces it with the order of work the factory is actually following as of 2026-11-19, and with an account of what became of each August prediction. The reader will learn what the next milestone is and what stands in front of it, what is planned after it, which large pieces of product code are sitting switched off on `main`, and what Yesod is trying not to become.

The plan is short at the near end and vague at the far end, and the chapter keeps that shape. Where a statement rests only on the working notes that the coordinating agents kept, it says so.

21.1. One Goal at a Time

Between 2026-08-20 and 2026-11-17 the repository gained 3,084 commits and grew from 186,926 to 535,696 lines of Python under `yesod/src/yesod` (`git rev-list --count --since=2026-08-20T00:00:00Z origin/main`, and `wc -l` at the two snapshots). Very little of that growth went toward the August chapter's ideas. It went into making the factory's own delivery mechanical, into a product shape in which a Yesod installation builds other people's repositories, and into the evidence and authority machinery both of those need.

The monthly counts show how the work then narrowed. September has 1,150 commits on `main` and October 1,400. November 1 to 17 has 109, almost all of them hardening the controller replacement path. Admissions were paused for most of the month around replacement windows, and everything not serving the first goal was parked. That reading of the drop is inferred from the counts and the agents' working notes for 2026-11-10.

The order of goals was posted to the factory's bulletin board on 2026-10-27, according to the mayor's notes, as five goals labeled G1 to G5. The notes for 2026-11-10 record the rule Stephen set for them: nothing on goals G2 to G5 until goal G1 has a published receipt.

21.2. Goal G1: One Automatic Swap

Goal G1 is "deploy on merge". Its test of completion, as Stephen stated it and the agents' working notes for 2026-11-10 record it, is one controller swap in which a successor machine is born, handed authority and serving, triggered by a merge, with a published receipt and "no hand-run step and no operator fix during the swap".

Chapter 11 (Deploy Is a Host Swap) describes the design and its status in full. The summary matters here because everything else waits on it.

> **Proven Where — as of 2026-11-19.** Designed: the whole merge-to-swap chain. On `main`: the chain is wired from the refinery's deployment step through `submit_replacement_intent` to the driver and the receipt. Rehearsed on disposable machines: successor birth and first-stage qualification, with blockers found and fixed; hand-submitted swaps, using reference code; the first birth of the built hardening code on a rehearsal box was on 2026-11-19 and produced further findings; the merge-triggered path has never run, even in rehearsal. Run in production: adoption of the hand-born serving controller (2026-11-10) and a green read-only dry run (2026-11-12). Not run in production: the handover. No replacement intent has reached `succeeded` there, and three attempts at a driver-born successor ended needing an operator. None reached the handover.

What stands between that status and goal G1, in the order the notes give it:

1. The hardening work called H1 has to land. It is one note with nineteen child tasks, on its fifth revision and still in its test gate on the evening of 2026-11-19, and it is not on `main`. Its children all closed without their database-backed tests having run (Chapter 7, Governed Agent Execution), so the evidence for it is the test run that the forensics agent performs on the delivered commit.
2. A database migration has to go through a hand-run window. The automatic path refuses any candidate that carries one: `ReplacementMigrationCoordinator.complete_automatic` raises `AutomaticMigrationRefused`, and the intent ends `needs_operator`. This is deliberate, and it means migrations will keep needing a human for now.
3. The handover has to complete once on machines away from production. Stephen required this after the mid-November attempts. Hand-submitted swaps have been rehearsed on disposable machines using reference code. The first birth of the built hardening code on a rehearsal box was on 2026-11-19 and is what exposed finding #136. A merge-triggered swap has never run anywhere.
4. A fourth production attempt has to pass birth and qualification, then the handover. The notes call these stages P6 and P7. The first pass is planned as a hand-started run from a release tree installed beside the serving one, under a signed grant (Chapter 18, Signed Human Authority).
5. Only then comes the swap that counts: the replacement driver's timer enabled on the serving controller, a merge, and no operator. As coded on `main` that swap cannot complete: the handover's quiet-gap proof counts the triggering merge's own attempt, waiting in `deploying`, as in flight (FINDING #132 of 2026-11-19, first read during the fact-check of this book and told in Chapter 11). The merge-triggered path has never run anywhere, including in rehearsal, so this step needs a code change and a rehearsal of its own. As of 2026-11-11 the notes record that timer and its service as deliberately disabled.

Step 5 is therefore the second successful swap, and the first has not happened.

There is a backlog waiting behind it. According to the agents' working notes, merges to Yesod's own repository currently end `merged_not_deployed`, including the one that produced the source snapshot this edition is written against. That merge is the candidate commit

`16606a42` (stamped 2026-11-17 19:16Z, a time derived from its source commit, not the time of landing); the working notes place the merge window on 2026-11-18. The notes for 2026-11-03 count 139 such changes carried by the serving controller. The first successful swap would deliver all of them at once, which is one more reason to rehearse it.

21.3. After G1: Goals G2 to G5

The remaining goals are known to this edition only in outline, from the mayor's notes of the 2026-10-27 bulletin post.

Goal	Content as recorded	Status
G2	"PLAN v2", step 1	waiting on G1
G3	"PLAN v2", step 2	waiting on G1
G4	Activate the dg6 gate fleet generation; retire dg5	tooling on `main`; live state not verified
G5	"PLAN v2", steps 3 and 4	waiting on G1

The content of "PLAN v2" was not available to the research for this edition, so the table gives step numbers and no more. G4 is concrete: dg5 and dg6 are gate fleet generations, the sets of nine gate machines described in Chapter 12 (Born, Not Repaired), and the lifecycle tooling for dg6 was committed in October (`2df252d65`).

One pressure will return as soon as merges flow again. By 2026-10-10 the agents' working notes name the gate, at about 70 minutes per candidate and one candidate at a time, as the limit on throughput, with runner seats idle behind it. The overflow capacity built for this, a Fargate Spot pool, exists on `main` as synthesis-only tooling whose runbook gives its status as "offline review tooling only", with the stack apply held as an operator action. Nothing on `main` records it having run.

21.4. Factories

The first project after goal G1 is a plan for several factories side by side. It is a plan only. According to the agents' working notes for 2026-11-15, no notes have been filed for it and nothing has started.

The plan has three parts: one network per factory; a naming scheme that carries the service, the factory and the environment; and a factory descriptor in code that replaces addresses hard-coded in scripts. A finding dated 2026-11-08 in the working notes had counted about twenty scripts that hard-code one network value.

The reason is the autumn's most expensive lesson. According to the working notes, a path that had never run end to end stopped in production in mid-November, and a rehearsal on a scratch machine then found further blockers. A second factory is meant to make that rehearsal

ordinary: a complete copy on which a swap can be run to completion before production sees it. It is also the shape of the product, since an installation at someone else's site is a factory with its own network and names.

The notes add that Stephen is porting parts of Yesod to Rust in parallel and wants a comparison after the network work. That work is outside the factory repository and this edition has not examined it.

21.5. The Hosted Product

The largest body of new code that this book does not otherwise describe is the hosted product. It is on `main`, it is dormant, and as far as `main` records it has never been installed.

The design is one Yesod installation per customer deployment: single-tenant, on AWS, with a thin client called `yc` (package `yesod-client`). The pieces on `main` are a hosted API runtime (`hosted_server.py`, `hosted_runtime.py`, `hosted_repositories.py` and nine sibling modules, about 13,300 lines), eleven database migrations, CDK stacks under `infra/aws_hosted/` with frozen synthesized templates in `infra/cdk.out.snapshot/`, operator tooling under `ops/aws-hosted/`, and the commands `yesod api bootstrap-owner` and `yesod api serve`.

Each of these landed through the same gate as everything else and changed nothing live, because each sits behind a switch that is off:

- The hosted API configuration defaults `enabled` to false in `hosted_config.py`, and `hosted_server.py` refuses to build the runtime without it, with the message "hosted API is disabled".
- The install runbook and the bootstrap procedure on `main` each open with a status line containing `live_install_status=not_run`.
- The Lambda MicroVM image runbook is marked "Status: HOLD".
- The managed PostgreSQL compatibility document says that, as checked on 2026-10-06, no live managed-service drill has occurred.

So the status is exact: designed, in code, proven by offline synthesis and tests that use fakes, and never deployed to an AWS account as far as anything on `main` at 2026-11-17 records. The working notes read for this edition do not claim a live install either. According to the working notes, rented AWS machines were used in October as scratch capacity for test runs of the home factory. That is a different thing and should not be mistaken for hosted Yesod.

Three decisions shape the program. The first, recorded on 2026-08-24, ended an earlier approach in which hosted features merged to a long-lived side branch. By then production had its own planning, execution and refinery authorities, and the contracts written against the branch proposed a second set of work controllers beside them. The decision calls the branch historical evidence only, never one to revive or merge into. The second, on 2026-10-12, fixed the first installable profile as `single_account_experiment`: one account, one region, CloudFormation as the only authority over deployed state, plain PostgreSQL 17, and the sentence "This profile is not production-certified." A cost model on `main` estimates the always-on control plane of that profile at 461.68 dollars a month, excluding variable costs (`ops/aws-hosted/cost/us-west-2-experiment.json`, priced 2026-10-12). The file's status is `review_required_before_live_apply`. It is a model, and no bill exists.

The third is the release plan in `docs/hosted/v1-release-gates.md`, which defines three milestones:

- **Beta** proves that one isolated tenant can be deployed, users can authenticate, and shared notes bind safely to the catalog.
- **Thin-factory** proves that one authorized project can deliver a note through planning, approval, execution, the gate and a fenced landing.
- **MVP** adds bounded projects, credential isolation, approval, budget and retention controls, and authorized promotion into an observed AWS deployment.

The document classifies every gate under each milestone as present, an adapter gap, a hosted gap or an AWS gap, and states the rule that a gate is green only when its exact required evidence exists. Nothing on `main` records any of the three milestones as declared.

According to the mayor's notes, the program paused twice behind the factory's own delivery: from 2026-09-16 to 2026-10-10 behind the gate's move to home hardware, and from late October behind goal G1. The notes for 2026-10-05 record "0 of 6 phases done" at that point. The commit history agrees on the gap and ends it a few days earlier: no commit on `main` dated between 2026-09-16 and 10-04 touches the hosted modules, `infra/`, `ops/aws-hosted/` or the client, and the hosted API entrypoint commits are dated 10-07 and 10-08. The densest week of product work was 2026-10-11 to 10-16, and no hosted or AWS commits landed in November apart from one fix that touched an AWS command-line module (`acf91372a`).

21.6. Two Deployment Targets

According to the agents' working notes for 2026-10-14, Stephen set two first-class targets for the product: AWS-hosted, and a version that users host themselves on their own hypervisor with local inference behind a model gateway. Parity means both.

This changes how the autumn's work should be read. If users will run Yesod on their own hardware, then the host swap and the opt-in, missing-only migration path described in Chapter 10 (Merged Is Not Deployed) and Chapter 11 are the product's upgrade path, and any state on the home hypervisor that cannot be rebuilt from the repository is a product defect. The same notes record Stephen calling Yesod building Yesod "a special case" that the design should not over-fit.

Other people's repositories already enter through a policy. `yesod onboard --github <url>` and the reviewed fleet policy file, `config/onboarding-fleet.yaml`, both date from 2026-08-09, before this period. The file now lists 27 managed and 8 excluded repositories, and since 2026-08-24 onboarding refuses a repository the policy does not list (`75f64a673`). On 2026-09-09 a forge-neutral contract (`forge.py`, `forge_gitlab.py`) added GitLab as a second backend. `config/refinery-lanes.json` defines 24 refinery lanes, each with its own repository checkout.

21.7. What the August Edition Predicted

The August chapter described ten directions and a near-term list. The table records each against `main` at `16606a42`; the first two directions share a row.

August prediction	What became of it	Status
Typed tool ecology; workflow compiler	Typed contracts arrived for the factory's own records only	Not built
Reciprocal tool improvement	Per-tool bulletin boards; no automatic filing	Partly, differently
Adaptive dispatch economist	Cost accounting landed; no router reads it	Not built
Factory digital twin	Real disposable copies: scratch machines, rehearsals	Changed direction
Reliability immune system	Separate alarms, reconcilers and observers	Partly
Signed provenance	Many bound receipts; no single attestation	Closer, another route
Multi-repository change sets	Each refinery lane still drains alone	Not built
Upstream contribution	Routed through the forge contract	Partly; path untraced
Living documentation	No sentinel exists	Not built
Near-term list	Gateway and second vendor staged; birth records	Mostly done

Three rows deserve a sentence more. The digital twin was imagined as a statistical replay of the run ledger. What the factory needed instead was a real copy to break, and that is what scratch machines and the planned second factory provide. Signed provenance was imagined as one cryptographic attestation from intent to runtime. What exists is a chain of receipts bound by digest: gate birth records, a run release attestation, birth and replacement receipts. Nothing in the repository verifies a human signature. The dispatch economist remains a stated want; the complexity estimator is unchanged in substance since June, and Chapter 16 (Cost-Aware Dispatch) describes what was built around it.

The living-documentation row has a footnote. This edition was produced by agents reading source read-only and checking each other's drafts, by hand, because the sentinel that would have flagged the August text as stale was never built.

Across the table, each time there was a choice between making the factory smarter and making it able to deliver and replace itself, the second came first, usually because a failure forced it.

21.8. What Yesod Should Not Become

The August list of negative requirements stands, and the autumn adds to it.

Yesod should not become an opaque swarm whose agents can change production because another agent said it was fine. It should not optimize one metric until every task looks like the metric. It should not let generated notes flood the catalog, erase history to keep a dashboard green, or let self-modification bypass the gate that ordinary work must pass.

It should not mistake discipline for enforcement. Most of the governance described in Part V is practice held by agents and one human, and the rule registry says so about itself. A future edition should be able to report that more of it moved into code, or that the rules which stayed attested were kept on purpose.

It should not become so careful that it stops. On 2026-11-11 every control worked, nothing unsafe happened, and one reviewed change took from the morning until the evening to merge, after a first window that merged nothing (Chapter 18). Safety bought with unbounded human round-trips does not scale.

It should not count dormant code as product. The hosted program is about thirteen thousand lines that have never served a request outside a test.

It should not confuse autonomy with the absence of human authority. A good factory reduces the number of decisions that need attention and makes the remaining ones better informed. One tension runs under all of this: letting an agent do a thing, or making the behavior deterministic code even when that code grows large. Yesod keeps choosing the code and reserving agents for the seams that cannot be decided mechanically. The autumn tested that choice hard, and the half-million lines are its price.

The name still fits the ambition. *Yesod* is Hebrew for foundation. The next thing the foundation has to do is replace itself once, on its own, with a receipt.

Source Notes

Claim or section	Tag	Anchor
Growth and monthly commit counts	[code]	`git rev-list --count --since=2026-08-20T00:00:00Z origin/main`, months by committer date in UTC; snapshots `c7a3e4d2`, `16606a42`
Goal order; G1 done-test; G2-G5 rule	[notes]	agents' working notes, 2026-10-27 and 2026-11-10
Proven Where; steps to G1	[code], [notes]	`routine_deployment.py`, `replacement_migration.py`, `replacement_cli.py`, `docs/runbooks/replacement-driver.md`; agents' working notes, 2026-11-10 to 11-19

Claim or section	Tag	Anchor
`merged_not_deployed` backlog	[notes]	agents' working notes, 2026-11-03 and 2026-11-18
Gate throughput; burst pool	[notes], [code]	agents' working notes, 2026-10-10; `docs/runbooks/aws-fargate-burst-pool.md`; `2df252d65`
Factories; Rust port	[notes]	agents' working notes, 2026-11-15
Hosted product and its status	[code], [deploy]	`hosted_config.py`, `hosted_server.py`; `docs/hosted/v1-release-gates.md`; `ops/aws-hosted/`; status lines of the hosted install, bootstrap, MicroVM image and managed PostgreSQL runbooks on `main`, 2026-11-17
Decisions of 08-24 and 10-12	[code]	`decisions/008-aws-hosted-production-factory-re-anchor.md`, `decisions/010-aws-hosted-cdk-experiment-profile.md`
Two targets; pauses	[notes], [code]	agents' working notes, 2026-10-05 and 2026-10-14; `git log` on the hosted and AWS paths; `acf91372a`
Onboarding and lanes	[code]	`config/onboarding-fleet.yaml`, `config/refinery-lanes.json`, `forge.py`, `forge_gitlab.py`; commits `75f64a673`, `817ecff68`, `db2945d98`
Predictions table	[code], [inferred]	research brief 12 section 7 against `16606a42`

Not verified:

- Whether anything AWS-hosted has ever run live. `main` says no in every runbook read, and the working notes read do not claim it; Stephen should confirm.
- The content of "PLAN v2" (goals G2, G3 and G5). Factories are the first project after goal G1; the book does not rank them against goal G2.
- The scope of H1 beyond what Chapter 11 reports from the notes.
- Whether a rehearsal of the handover away from production has reached `succeeded`.
- The three production attempts of mid-November and the findings of the evening of 2026-11-19 are told in Chapter 11 from the mayor's notes and were not otherwise verified.
- Whether the upstream-contribution finish path runs on the deterministic controller path; whether retention cleanup is wired.
- The factories plan document itself was not read; the account is from the working notes.
- Counts from the working notes (139 undeployed changes, about 70 minutes per gate run, about twenty hard-coded scripts) were not measured from source.

Epilogue: Scar Tissue

Many systems can ask an LLM to edit code. That part of Yesod is ordinary.

The part worth studying is the accumulation of refusals.

Do not call an exit code a commit. Do not call a local commit a remote branch. Do not call a branch a merge. Do not call a merge a deployment. Do not call a transport failure a red test. Do not call an unknown price zero. Do not call an agent's claimed identity authority. Do not retry forever. Do not launch forever. Do not let one blocked row stop unrelated work. Do not let a watchdog kill a healthy gate. Do not let a process touch a checkout it does not own.

The autumn added its own.

Do not call a closed child a tested change. Do not call a merged commit a running one. Do not call a green dry run a rehearsal. Do not call a review an execution. Do not call a summary the thing that was signed. Do not treat a timeout as a yes. Do not treat silence from a monitor as good news. Do not borrow someone else's pause. Do not migrate a database because a process happened to start. Do not restart a process that is holding work. Do not repair a host that could be born again. Do not roll back past the point where rollback stopped being safe.

Each refusal is scar tissue from a failure that once looked plausible.

Agents are cheap to add. What makes an orchestrator a factory is the number of important claims it knows how to prove, and the honesty to say which ones it cannot prove yet.

As of 2026-11-19 there is one such claim outstanding above all others. The automatic, merge-triggered swap of the controller is designed and on `main`, and it has not completed in production: three attempts at a driver-born successor ended needing an operator, and none reached the handover. On the evening of that day the hardening for a fourth attempt met seven more findings in rehearsal, before any production step, and one of them is that the merge-triggered swap deadlocks as coded. The swap is not done.

A Command Field Guide

This appendix lists the `yesod` commands a reader is most likely to need, grouped by purpose. Every command and flag printed here was checked against the command definitions at `origin/main` `16606a42` (2026-11-17). It is a selection. The command line has many more verbs than fit on these pages, and `--help` on any group shows the rest.

Each group has two blocks. The first holds commands that only read state. The second is labelled "mutating: governed by the authority rules in Chapter 18" and holds commands that change notes, queues, leases, hosts or mail. Chapter 18 (Signed Human Authority) says who may run those and under what grant. A flag can move a command from one block to the other: `--dry-run` on a mutating verb makes it a check, and `--apply` on a reporting verb makes it a write. Where that happens the line says so. "Read-only" means the verb does not change factory state. It does not mean no row is written: the root callback records every invocation with a subcommand in the `yesod_invocations` table (`_log_invocation` in `cli.py`), best-effort.

Placeholders are in angle brackets. `<tool>` is the registered tool name (for the factory itself, `yesod`). `<note-id>` is a note id such as `ys-yes-4b16`. Most reporting verbs accept `--json` for machine output and `--pretty` for a table.

> **Terminology Trap — two things called "status".** `yesod codefactory-dispatch status` reports runners and the dispatch queue. `yesod refinery status` reports the merge pipeline. `yesod factory status` reports the agent seats. None of them reports what release a host is running; that is `yesod release-skew`.

A.1. Catalog and Knowledge

Read-only:

```
yesod version --json                # release identity of this CLI
yesod tools list                    # registered tools, by kind
yesod tools count
yesod search "<term>"               # tools and notes by keyword
yesod search "<term>" --semantic    # embedding search
yesod ask "<question>"              # cited answer from the catalog
yesod ask "<question>" --json
yesod query <tool> --question "<question>"
yesod tool <tool> notes --detail    # notes filed on one tool
yesod tool <tool> notes --kind <kind> --active
yesod topics list
```

```
yesod topics show <topic>
yesod topic <topic> notes
yesod prime                          # agent-facing usage guide
yesod prime --worker                 # role primer (one flag per role)
```

Mutating: governed by the authority rules in Chapter 18.

```
yesod tool <tool> usage-note "<text>" --agent <name>
yesod topic <topic> usage-note "<text>" --agent <name>
yesod tool <tool> note-comment <note-id> "<text>"
yesod tool <tool> supersede <note-id> --with "<replacement text>"
yesod tools add <name> --kind <kind> --description "<text>"
yesod render-agents-md               # rewrite the canon block
yesod render-agents-md --check       # read-only: fail if it is stale
```

`supersede` keeps the old note and links the replacement, which is the right choice when a fact changed. `yesod tools add` registers a new tool; registration is the owner's decision.

A.2. Notes and Lifecycle

Read-only:

```
yesod notes list --tool <tool> --status <status>
yesod notes list --active --json
yesod notes show <note-id> --full
yesod notes dep list <note-id>
yesod notes dep integrity
yesod note-status-history <note-id> # every status change, with actor
yesod note-watch <note-id>          # block until the note moves
yesod why-not-running <note-id>     # evidence for a stuck note
yesod tool <tool> note-why <note-id>
yesod tool <tool> note-inspect <note-id> --json
yesod tool <tool> note-plan-status <note-id>
yesod tool <tool> note-attachments <note-id>
yesod tool <tool> note-admin <note-id> --show
```

`note-why` and `note-inspect` report the note's status, disposition, plan binding and the first reason it is not eligible for the next step. `why-not-running` adds runner and routing evidence.

Mutating: governed by the authority rules in Chapter 18.

```
yesod tool <tool> feature-request "<text>" --agent <name>
yesod tool <tool> bug-report "<text>" --agent <name>
yesod tool <tool> feature-request "<text>" --depends-on <note-id>
```

```
yesod tool <tool> note-capture <note-id>
yesod tool <tool> note-request-planning <note-id>
yesod tool <tool> note-plan <note-id>
yesod tool <tool> note-approve <note-id>
yesod tool <tool> note-hold <note-id> --reason "<why>"
yesod tool <tool> note-unhold <note-id> --reason "<why>"
yesod tool <tool> note-release <note-id> --reason "<why>"
yesod tool <tool> note-rework <note-id> --reason "<why>"
yesod tool <tool> note-quarantine <note-id> --reason "<why>"
yesod tool <tool> note-depends <note-id> --on <other-note-id>
yesod tool <tool> note-edit <note-id> --body "<text>" \
  --if-revision <n>
yesod tool <tool> note-status <note-id> <status>
```

`note-plan` is a writing verb despite its name: it requests revision-bound planning and stops at `planned` for approval. `note-request-planning` is an alias for it. Use `note-plan-status` to look. `note-hold`, `note-unhold`, `note-release`, `note-rework` and `note-quarantine` change the note's disposition and leave its lifecycle status where it was (Chapter 5, Capture, Plan, Deliver). `note-edit` with `--if-revision` refuses if the body changed since the revision named. `note-approve` approves a current validated plan and promotes the note from `planned` to `open`.

A.3. Beads Routing

All Beads work goes through `yesod bd <tool> -- <bd arguments>`, which selects the tool's workspace and connection. Everything after -- is passed to `bd` unchanged.

Read-only:

```
yesod bd <tool> -- show <bead-id>
yesod bd <tool> -- children <bead-id>
yesod bd <tool> -- dep list <bead-id>
yesod bd <tool> -- graph <bead-id>
yesod bd <tool> -- worktree info
yesod beads doctor                  # workspace and connection check
yesod beads info <tool>
yesod beads ready <tool> --json
yesod beads fleet-status
yesod beads provenance <tool> <bead-id>
yesod beads ref <reference>         # resolve a note or bead reference
```

Mutating: governed by the authority rules in Chapter 18.

```
yesod bd <tool> -- ready --claim --json
yesod bd <tool> -- create --parent <bead-id> --type task \
  --title "<title>" --priority P2
```

```
yesod bd <tool> -- note <bead-id> "<text>"
yesod bd <tool> -- comment <bead-id> "<text>"
yesod bd <tool> -- assign <bead-id> <actor>
yesod bd <tool> -- close <child-bead-id>
yesod bd <tool> -- worktree create <path> --branch <name>
yesod bd <tool> --source-root <checkout> -- \
  worktree create <path> --branch <name>
```

`ready --claim` picks the next ready child and claims it in one step, so it writes. `--source-root` is consumed by Yesod and never reaches `bd`; it pins the checkout a worktree is cut from when detection is ambiguous. `--subprocess-timeout-seconds` is the other routing option accepted before the `--`. Workers close child beads only; the root bead belongs to the runner and the refinery (Chapter 7, Governed Agent Execution).

A.4. Dispatch and Runner

The daemon behind `yesod codefactory-dispatch` is the runner.

Read-only:

```
yesod codefactory-dispatch status --pretty
yesod codefactory-dispatch queue --pretty
yesod codefactory-dispatch queue --held
yesod codefactory-dispatch logs --note <note-id>
yesod codefactory-dispatch routing-log --note <note-id>
yesod codefactory-dispatch runner list --pretty
yesod codefactory-dispatch runner drains
yesod codefactory-dispatch planning --note <note-id>
yesod codefactory-dispatch planner-fleet
yesod codefactory-dispatch lifecycle-v2 status
yesod planning-queue --pretty       # every active planning job
yesod runs list --note <note-id>
yesod runs list --status <status> --since <when>
yesod runs show <run-id>
yesod runs failures
yesod runs logless
yesod runs rollup <root-bead-id>
yesod runs planning-report
yesod daemon-sweep                  # daemons found on this host
```

Mutating: governed by the authority rules in Chapter 18.

```
yesod codefactory-dispatch arm <note-id> \
  --target <lane> --reason "<why>"
yesod codefactory-dispatch replan <note-id>
```

```
yesod codefactory-dispatch planning-retry <job-id>
yesod codefactory-dispatch planning-cancel <note-id>
yesod codefactory-dispatch planning-reconcile \
  --note <note-id> --dry-run       # read-only with --dry-run
yesod codefactory-dispatch planning-reconcile \
  --note <note-id> --apply
yesod codefactory-dispatch planner-fleet --max-concurrent <n>
yesod codefactory-dispatch planned-root-retry <note-id>
yesod codefactory-dispatch retained-child-reconcile <note-id> \
  --run-id <run-id> --result-commit <sha> --worktree <path>
yesod codefactory-dispatch runner drain <runner-id> \
  --reason <drain-type> --expires <duration> --actor <actor>
yesod codefactory-dispatch runner resume <runner-id> \
  --reason "<why>" --actor <actor>
yesod codefactory-dispatch sweep-stale --dry-run
yesod runs kill <run-id>
yesod dispatch-agent-log <note-id> "<one-line summary>"
```

`arm` sets the dispatch lane (the model target) for a note. `runner drain` takes a drain type for `--reason` (`maintenance`, `rollout-prep`, `incident` or `host-work`) and a lease between 15 minutes and 24 hours for `--expires`. `retained-child-reconcile` reports what it would do until `--apply` is added. `runs kill` records a kill request that the runner's governance loop acts on at its next check. `dispatch-agent-log` is the worker heartbeat; it writes a row to `dispatch_log` and is safe for any worker to call.

A.5. Refinery and Admissions

Read-only:

```
yesod refinery status --pretty      # the merge pipeline at a glance
yesod refinery list --pretty --full
yesod refinery show <queue-id>
yesod refinery check-stale
yesod refinery lock
yesod refinery workers
yesod refinery profile list
yesod refinery profile show <tool>
yesod refinery settle --dry-run     # settlement actions, not applied
yesod refinery admissions status --json
yesod lane show --lane <lane-id> --pretty
yesod lane attempt <attempt-id>
yesod lane conflicts --note <note-id>
yesod lane ordinals --note <note-id>
yesod lane audit <target>
yesod main-ci status                # latest recorded main CI result
```

```
yesod main-ci list
```

`refinery admissions status` prints the `paused` flag and the `revision` that the pause and resume verbs must be given. `lane` here is the refinery lane (a repository and target branch). The two `main-ci` verbs call `init_db()` before reading. On `main` that call only verifies the schema version unless the process sets `YESOD_ALLOW_MIGRATE=1`, in which case it can migrate; run them without that variable.

Mutating: governed by the authority rules in Chapter 18.

```
yesod refinery admissions pause --actor <actor> \
  --reason "<why>" --expect-revision <n> \
  --token-out <file> --confirm PAUSE
yesod refinery admissions resume --actor <actor> \
  --reason "<why>" --expect-revision <n> \
  --pause-token-file <file> --confirm RESUME
yesod refinery enqueue --note <note-id> --strict
yesod refinery submit-branch --lane <lane-id> \
  --branch <branch> --source-sha <sha> --bead <bead-id> \
  --repo-dir <checkout> --actor <actor> \
  --dry-run                          # checks, then rolls back
yesod refinery rearm --note <note-id> \
  --repo-dir <checkout> --reason "<why>"
yesod refinery attempt cancel <attempt-id> --reason "<why>"
yesod refinery dequeue <queue-id>
yesod refinery inbox acknowledge <inbox-id> --actor <actor>
yesod refinery inbox resolve <inbox-id> --actor <actor> \
  --reason "<why>"
yesod refinery lanes ensure --tool <tool> \
  --repository <repo> --target-branch <branch> \
  --expect-lane-id <lane-id> \
  --expect-admission-revision <n> --confirm ENSURE
```

The pause writes a token file; only the holder of that file and the matching revision can resume (Chapter 8, The Merge Controller). `enqueue --strict` refuses while admissions are paused. `lanes ensure` registers a refinery lane and is accepted only while admissions are paused. `set-status`, `set-order` and `bypass-blocked` also exist on `yesod refinery`; they edit refinery queue rows directly and are left out of the blocks above.

A.6. Gate

“Gate” here is the merge-time test gate.

Read-only:

```
yesod refinery gate-report <attempt-or-note-id>
```

```
yesod refinery gate-report <note-id> --all --json
yesod refinery gate-report <attempt-id> \
  --show-failure-tracebacks
yesod refinery gate-plan --json
yesod refinery gate-env --lane <lane-id>
yesod gate-db-health
yesod fleet gate-credential list
yesod fleet gate-credential status <credential-name>
yesod fleet gate-credential audit-log
yesod refinery fargate-cleanup \
  --installation-config <file> --json  # report only
```

Mutating: governed by the authority rules in Chapter 18.

```
yesod refinery abort-gate <queue-id> --reason "<why>"
yesod fleet gate-credential set <credential-name> --file <path>
yesod fleet gate-credential rotate <credential-name> --file <path>
yesod fleet gate-credential refresh
yesod refinery fargate-cleanup \
  --installation-config <file> --apply
```

The credential verbs take the value from a file, never from the command line. Do not paste a credential into a terminal, a note or a mail message.

A.7. Release, Skew and Replacement

Read-only:

```
yesod release-skew                  # release identity per surface
yesod release-skew --json --strict
yesod release-skew --local-only
yesod version --expect-release-sha <sha>
yesod replacement status <intent-id>
yesod replacement drive --dry-run   # readiness checks, no effects
yesod release-promotion status <promotion-id>
yesod fleet status --pretty
yesod fleet stop-check
```

`release-skew` compares every surface (operator CLI, controller, runner seats) against a Git reference; `--strict` exits non-zero when any reported surface is not current. `replacement drive --dry-run` reports every readiness check without driving effects. As of 2026-11-19 a fully automatic, merge-triggered controller swap has not completed in production (Chapter 11, Deploy Is a Host Swap), so a green dry-run is evidence of readiness and nothing more.

Mutating: governed by the authority rules in Chapter 18.

```
yesod replacement submit --candidate-sha <sha> --from-binding
yesod replacement drive --stop-before-handover
yesod replacement drive
yesod replacement adopt --actor <actor> --reason "<why>" \
  --expected-admission-revision <n> \
  --pause-token-file <file> --lease-seconds <n> --dry-run
yesod replacement adopt --actor <actor> --reason "<why>" \
  --expected-admission-revision <n> \
  --pause-token-file <file> --lease-seconds <n>
yesod replacement publish-receipt <path>
yesod release-promotion hold <promotion-id> --reason "<why>"
yesod release-promotion resume <promotion-id>
yesod fleet all-stop <address>
yesod fleet all-stop --all
yesod fleet all-stop --all --clear
```

`replacement drive` runs one restartable pass of the driver; the timer unit on `main`, `yesod-replacement-driver.timer`, starts it once a minute. `--stop-before-handover` births and qualifies a successor without entering handover. `adopt --dry-run` collects precondition failures without writing state; `--adoption-candidate <file>` gives it an uninstalled adoption document to check. Exit codes for `yesod replacement` are 0 succeeded, 1 error, 2 rolled back, 3 needs operator and 75 not yet terminal (`EXIT_*` in `replacement_cli.py`).

A.8. Mail, Bulletin and Agents

Read-only:

```
yesod mail whoami
yesod mail roster --pretty
yesod mail check <address>
yesod mail check <address> --wait --timeout <seconds>
yesod mail thread <root-id> --pretty
yesod bulletin list <topic> --since <when> --limit <n>
yesod bulletin read <post-id>
yesod bulletin board list
yesod bulletin board role list <board>
yesod bulletin config show
yesod factory status
yesod fleet agents
yesod agent roster --pretty
yesod agent list --live
yesod agent inspect <address>
```

`mail check` lists unread messages without marking them, but it does update the caller's last-active time on the roster (`mail_store.bump_active`). `mail read <message-id>` shows one message and marks it read, so it is listed below.

Mutating: governed by the authority rules in Chapter 18.

```
yesod mail register --role <role>
yesod mail read <message-id>
yesod mail send <address> --subject "<subject>" \
  --message "<text>"
yesod mail reply <message-id> --message "<text>"
yesod mail ack <message-id>
yesod mail ask "<question>" --deadline <when> \
  --choices "<a,b>"
yesod bulletin post <topic> --kind <kind> \
  --title "<title>" --message "<text>"
yesod bulletin comment <post-id> "<text>"
yesod bulletin promote <post-id> --synthesis "<text>"
yesod bulletin board role set <board> <selector> <role>
yesod fleet deliver <address> "<text>" --subject "<subject>"
yesod agent spawn --role <role> --prompt "<text>"
yesod agent deactivate <address>
```

`mail ask` puts a blocking question to Stephen through the human channel and returns the answer or a recorded timeout; `--choices` is a comma-separated list. The signing tool that carries his permission is separate and outside this repository (Chapter 18).

A.9. Rules

Read-only:

```
yesod rules check                   # evaluate every rule
yesod rules check <rule-key> --since <when> --json
yesod rules show <rule-key>
yesod rules show <rule-key> --show-query
```

`yesod rules` has no mutating verbs. It evaluates the compliance rules in `rules_contract.RULES` against recorded evidence and reports; it does not enforce anything (Chapter 17, The Agent Organization).

A.10. Scratch Machines

Read-only:

```
yesod scratch list --mine
yesod scratch list --all
```

Mutating: governed by the authority rules in Chapter 18.

```
yesod scratch create --kind <kind> --ttl <duration> \
  --purpose "<why>"
yesod scratch extend <lease> --ttl <duration>
yesod scratch run-tests <lease> --tree <path>
yesod scratch destroy <lease>
```

A scratch lease is a disposable test machine with a time limit. Lease limits are listed in the States and Timers appendix.

Source Notes

Section	Tag	Anchor
Catalog and knowledge	[code]	`cli.py`, `ask.py`, `search.py`, `query.py`, `tools.py`, `topic.py`, `prime.py`, `worker_canon.py`
Notes and lifecycle	[code]	`tool.py`, `notes.py`, `note_status_history.py`, `note_watch_cli.py`, `diagnostics.py`
Beads routing	[code]	`beads_cli.py` (`bd_command`); `bd` arguments from the worker canon in `worker_canon.py` and `AGENTS.md`
Dispatch and runner	[code]	`dispatch_cli.py`, `runs_cli.py`, `planning_status_cli.py`, `daemon_sweep.py`, `dispatch_agent_log.py`
Refinery and admissions	[code]	`refinery_cli.py`, `lane_cli.py`, `lane_conflicts_cli.py`, `lane_ordinals_cli.py`, `lane_audit_cli.py`, `main_ci_cli.py`
Gate	[code]	`refinery_cli.py`, `fleet_cli.py` (`gate_credential_app`), `diagnostics.py`
Release, skew, replacement	[code]	`cli.py` (`release-skew`, `version`), `replacement_cli.py`, `release_promotion_cli.py`, `fleet_cli.py`
Mail, bulletin, agents	[code]	`mail_cli.py`, `bulletin_cli.py`, `agent_cli.py`, `factory_status.py`
Rules, scratch	[code]	`rules_cli.py`, `scratch_cli.py`

All files are under `yesod/src/yesod/` at `origin/main` `16606a42`. Commands were verified by reading the Typer decorators and option declarations; none was run.

Not verified:

- The arguments after `--` in `yesod bd` lines belong to the separate `bd` binary. They are printed as the worker canon on `main` gives them and were not checked against `bd` itself.

- Accepted value formats for `--since`, `--ttl`, `--deadline` and `--kind` were not traced; see `--help` for each verb.
- Whether each read-only verb is free of incidental writes (caches, heartbeats, an implicit schema check on connect) was not traced end to end. The classification is by the verb's purpose and its option declarations.
- `yesod info`, `yesod main-ci status`, `yesod main-ci list` and the exact-candidate form of `yesod version` (`--json` with a full 40-character `--expect-release-sha`) call `init_db()`. On `main`, `init_db` verifies the schema version without running DDL unless `YESOD_ALLOW_MIGRATE=1` is set (and `YESOD_SKIP_INIT_DB` is not). `yesod info` is omitted from the read-only block for that reason. A direct `init_db()` call was searched for in the other command modules listed below and not found; deeper call paths were not traced.
- How `_log_invocation` behaves when the database is unreachable or read-only was not traced beyond its `try` block.
- `yesod replacement plan` appears in the replacement-driver runbook but no such verb is registered in `replacement_cli.py`, so it is omitted.

B States and Timers

This appendix is a lookup table for the words and numbers the factory uses to describe itself. The first half lists each state vocabulary with its exact spelling and the symbol that defines it. The second half lists timers, limits and budgets with their constant names, values and files. Every name and value was read from source at `origin/main` `16606a42` (2026-11-17). File names are relative to `yesod/src/yesod/` unless a longer path is given.

Two cautions apply to every table. A code default is what the code uses when configuration says nothing; the running controller reads a reviewed configuration document that overrides several of them, and the tables say which value is which. And a state name describes a row in the database, not a fact about the world: `merged_not_deployed` means the controller recorded a merge and could not confirm a deployment, which is the subject of Chapter 10 (Merged Is Not Deployed).

B.1. Note Lifecycle Status

Defined by `IMPLEMENTATION_LIFECYCLE` in `note_lifecycle.py`, lifecycle contract version 2 (`CONTRACT.version`). Twelve states, stored in `tool_notes.status`. Feature requests and bug reports use this vocabulary.

Status	Group	Meaning
`captured`	intake	filed, not yet planned
`planning`	intake	a planning job is working on it
`planned`	intake	a validated plan awaits approval
`open`	intake	approved, eligible for dispatch
`claimed`	execution	a runner has claimed it
`in_progress`	execution	a worker run is active
`awaiting_verification`	delivery	a result exists and needs triage
`awaiting_merge`	delivery	a verified branch waits for the gate

Status	Group	Meaning
`merging`	delivery	gate green, merge being finalized
`done`	terminal	landed, or accepted with no code
`wontfix`	terminal	closed by decision
`superseded`	terminal	replaced by another note

Usage notes use `USAGE_LIFECYCLE` in the same file: `current`, `stale`, `superseded`.

> **Terminology Trap — `blocked` is not a status.** Earlier editions listed `blocked` among the note statuses. On `main` it is a disposition (next table) and also a planning job state, and it is absent from `IMPLEMENTATION_LIFECYCLE`. A note can be `in_progress` and `blocked` at once. Old rows can still carry the raw status `blocked`; a comment in `is_terminal` (`note_status.py`) says readers may meet it and treat it as not terminal.

B.2. Note Disposition

Defined by `DISPOSITIONS` in `disposition.py`. The disposition says whether a note is in play at all, independently of how far its work has gone. Every value except `active` vetoes dispatch, gating and merging (`NON_ACTIVE_DISPOSITIONS`).

Disposition	Constant	Set by
`active`	`ACTIVE`	default (`DEFAULT_DISPOSITION`)
`held`	`HELD`	`note-hold`
`blocked`	`BLOCKED`	automation (not traced)
`needs_rework`	`NEEDS_REWORK`	`note-rework`, refinery
`quarantined`	`QUARANTINED`	`note-quarantine`
`deferred`	`DEFERRED`	not traced

The "set by" column names the usual writer and was not traced for every path. The writer is `write_disposition` in `note_status_writer.py`, which records each change in `note_disposition_changes`; whether every path uses it was not traced.

Two neighboring vocabularies are short enough to give in a sentence. The automation policy of a note is one of `capture`, `plan`, `deliver` (`AUTOMATION_POLICIES`, `note_lifecycle_ops.py`). The dispatch column `agent_dispatch` holds a dispatch lane name, nothing, or the literal `hold` (`HOLD_DISPATCH_VALUE`, `db.py`).

B.3. Planning Job State

Defined in `planning_jobs.py`; rows live in `planning_jobs`, with history in `planning_job_events`.

State	Set	Meaning
`queued`	`ACTIVE_STATES`	waiting for a planner lease
`running`	`ACTIVE_STATES`	leased to a planner
`retrying`	`ACTIVE_STATES`	failed, will run again
`blocked`	`ACTIVE_STATES`	fenced, needs a decision
`succeeded`	`TERMINAL_STATES`	plan published and bound
`failed`	`TERMINAL_STATES`	ended without a plan
`cancelled`	`TERMINAL_STATES`	cancelled, or deadline passed
`superseded`	`TERMINAL_STATES`	a newer revision replaced it

Only `queued` and `retrying` can be claimed (`CLAIMABLE_STATES`).

A terminal job carries a `terminal_reason` JSON object whose `code` field says why. The codes below were found in source; the list is not exhaustive.

Reason code	Where it is set
`retry_exhausted`	`planning_jobs.py`, `planning_execution.py`
`deadline_exceeded`	`planning_execution.py`
`invalid_provider_output`	`planning_execution.py`, `planning_jobs.py`
`provider_start_failed`	`planning_execution.py`
`completed_prefix_invalid`	`planning_execution.py`
`bead_publish_failed`	`planning_publication.py`, `planning_execution.py`
`stale_note_revision`	`planning_publication.py`

Reconciliation code	Constant
`reconcile_noncurrent_revision`	`RECONCILE_NONCURRENT_REVISION`
`reconcile_note_terminal`	`RECONCILE_NOTE_TERMINAL`
`reconcile_disposition_fenced`	`RECONCILE_DISPOSITION_FENCED`

A reaped lease is recorded as an event with code `lease_expired` and moves the job from `running` to `retrying`; it is not a terminal reason.

B.4. Run Status

Defined in `run_ledger.py`; rows live in `runs`.

Status	Set
`starting`	`LIVE_RUN_STATUSES`
`running`	`LIVE_RUN_STATUSES`
`success`	`TERMINAL_STATUSES`
`failure`	`TERMINAL_STATUSES`
`killed`	`TERMINAL_STATUSES`
`timeout`	`TERMINAL_STATUSES`
`lost`	`TERMINAL_STATUSES`
`token_budget`	`TERMINAL_STATUSES`

B.5. Child Outcomes

A plan-bound child that ends without a commit reports one structured outcome in a heartbeat whose summary starts with `YESOD_CHILD_OUTCOME:` (`CHILD_OUTCOME_HEARTBEAT_PREFIX`, `child_outcomes.py`). The enums are in `contracts/planned_root.py`.

Outcome	Enum member
`blocked_on_dependency`	`ChildOutcomeKind.BLOCKED_ON_DEPENDENCY`
`no_changes_needed`	`ChildOutcomeKind.NO_CHANGES_NEEDED`
`acceptance_not_executed`	`ChildOutcomeKind.ACCEPTANCE_NOT_EXECUTED`

Not-executed reason	Enum member
`db_env_unset`	`AcceptanceNotExecutedReason.DB_ENV_UNSET`
`zero_collected`	`AcceptanceNotExecutedReason.ZERO_COLLECTED`
`exit_5`	`AcceptanceNotExecutedReason.EXIT_5`
`all_skipped`	`AcceptanceNotExecutedReason.ALL_SKIPPED`

A dependency named by `blocked_on_dependency` has kind `note`, `branch` or `bead` (`DependencyKind`).

B.6. Refinery Attempt State

Defined by `AttemptState` in `refinery_state.py`; rows live in `refinery_attempts`. A terminal attempt never reopens. A retry is a new attempt.

Active state	Meaning
`requested`	recorded, not yet claimed
`claimed`	the lane controller holds it
`preparing`	building the candidate commit
`gating`	the gate is running
`gate_green`	the gate passed
`pushing`	pushing to the target branch
`deploying`	waiting for deployment evidence

These seven form `ACTIVE_ATTEMPT_STATES`.

Terminal state	Meaning
`succeeded`	merged and confirmed deployed
`merged_not_deployed`	merged, deployment not confirmed
`gate_red`	the gate failed
`needs_rework`	the source work must change
`needs_operator`	automation cannot decide
`push_raced`	the target moved during push
`infrastructure_failed`	the machinery failed, not the code
`timed_out`	a deadline passed
`cancelled`	cancelled by request
`superseded`	replaced by a newer attempt

These ten form `TERMINAL_ATTEMPT_STATES`. `merge_conflict` is a reason code on an attempt, not a state.

B.7. Gate Execution State

Defined by `GateExecutionState` in `refinery_state.py`; rows live in `refinery_gate_executions`. "Gate" is the merge-time test gate.

State	Set
`submitted`	`ACTIVE_GATE_EXECUTION_STATES`
`starting`	`ACTIVE_GATE_EXECUTION_STATES`
`running`	`ACTIVE_GATE_EXECUTION_STATES`
`green`	`TERMINAL_GATE_EXECUTION_STATES`
`red`	`TERMINAL_GATE_EXECUTION_STATES`

State	Set
`timed_out`	`TERMINAL_GATE_EXECUTION_STATES`
`cancelled`	`TERMINAL_GATE_EXECUTION_STATES`
`lost`	`TERMINAL_GATE_EXECUTION_STATES`

A gate execution has a kind, `full` or `retest` (`GateExecutionKind`).

The pre-gate, a test run on scratch capacity before the real gate, has its own small vocabularies in `refinery_pre_gate.py`.

Enum	Values
`PreGateMode`	`off`, `shadow`, `active`
`PreGateState`	`requested`, `running`, `passed`, `failed`, `timed_out`, `unavailable`, `skipped`
`PreGateVerdict`	`green`, `red`, `unknown`

B.8. External Effect Evidence (Refinery)

Defined by `ExternalEffectEvidence` in `refinery_state.py`. It records what the controller knows about an action outside the database, such as a push.

Evidence	Meaning
`not_requested`	nothing was attempted
`intent_recorded`	about to act, outcome not yet seen
`outcome_unknown`	acted, cannot tell whether it applied
`confirmed_applied`	observed as applied
`confirmed_not_applied`	observed as not applied

`ExternalEffectAction` names what the controller may do next: `execute`, `reconcile`, `accept`.

B.9. Admissions State

There is no enum. The single row of `refinery_admission_control` (scope `global`) is read as an `AdmissionState` in `refinery_admission.py`.

Field	Type	Meaning
`paused`	boolean	new attempts are refused when true
`revision`	integer	increments on each change
`pause_token`	text	present only while paused
`actor`, `reason`	text	required while paused

A table constraint requires a token, a reason, an actor and a pause time whenever `paused` is true, and no token when it is false. `require_admissions_open` raises `AdmissionPaused` when the flag is set.

B.10. Replacement Intent Phase

Defined by `ReplacementPhase` in `replacement_contract.py` with the same edges in `_PHASE_TRANSITIONS` and in the transition trigger `trg_replacement_intents_fenced` on `replacement_intents` (migration `0117`). Thirteen phases. Every non-terminal phase may also move to `needs_operator`; from `fencing` onward a phase may move to `rolled_back`.

Phase	Next on the forward path
`planned`	`birthing`
`birthing`	`qualified`
`qualified`	`ready`
`ready`	`pausing`
`pausing`	`draining`
`draining`	`fencing`
`fencing`	`activating`
`activating`	`rolling_runners` or `settling`
`rolling_runners`	`settling`
`settling`	`succeeded`

Terminal phase	`TerminalDisposition`
`succeeded`	`SUCCEEDED`
`rolled_back`	`ROLLED_BACK`
`needs_operator`	`NEEDS_OPERATOR`

An intent's `deployment_kind` is `controller` or `runner_rollout`.

Proven Where — as of 2026-11-19. These phases are in code and in migrations on `main`. No replacement intent has reached `succeeded` in production. Three production attempts at a driver-born successor ended needing an operator, and none reached the handover, according to the agents' working notes (Chapter 11, Deploy Is a Host Swap).

B.11. Replacement Target State

Defined by `ReplacementTargetState` in `replacement_contract.py`; rows live in `replacement_targets`, one per host being born or retired. A target's role is `controller` or `runner`.

State	Next
`planned`	`born`
`born`	`qualified`
`qualified`	`passive`
`passive`	`serving`, or `stopped_retained`
`serving`	`stopped_retained`
`stopped_retained`	`destroyed`, or back to `serving`
`destroyed`	terminal
`failed`	terminal

Every state before `destroyed` may also move to `failed`. Rows in `destroyed` or `failed` are immutable. The edges are in `_TARGET_TRANSITIONS` and in the trigger function `replacement_targets_fence`, created in migration `0117` and replaced in `0119`, which added the edge from `passive` to `stopped_retained`.

B.12. Replacement Effect-Journal Evidence

The column `evidence_state` on `replacement_effect_journal`, constrained by `replacement_effect_journal_state_check` (migration `0117`). The journal has one row per external effect the replacement driver intends.

Evidence state	May move to
`intended`	any of the three below
`outcome_unknown`	either confirmed state
`confirmed_applied`	nothing (immutable)
`confirmed_not_applied`	nothing (immutable)

This vocabulary differs from the refinery's by one word: the journal says `intended` where `ExternalEffectEvidence` says `intent_recorded`, and it has no `not_requested`.

B.13. Replacement Stop-Guard State

The column `state` on `replacement_stop_guards`, one row per intent, constrained by `replacement_stop_guards_state_check` (migration `0117`). No Python enum was found for it.

State	Counted as live
`acquiring`	yes
`active`	yes
`proved`	yes
`blocked`	no
`released`	no
`expired`	no

"Counted as live" follows the partial index on the table, which covers `acquiring`, `active` and `proved`. The meaning of each state was not traced beyond the constraint; Chapter 11 describes what the guard is for.

B.14. Lifecycle Rollout Stage

`STAGES` in `lifecycle_rollout.py`: `disabled`, `readers`, `writers`, `planner`, `canary`, `active`, `rollback`.

B.15. Runner Timers

All in `runner.py`. These are code defaults; a runner's configuration file can override most of them through `RunnerConfig`.

Constant	Value	Purpose
`DEFAULT_POLL_INTERVAL`	10 s	claim loop cadence
`DEFAULT_HEARTBEAT_INTERVAL`	30 s	runner heartbeat
`DEFAULT_DEAD_THRESHOLD`	300 s	runner considered dead
`DEFAULT_LOOP_LIVENESS_BOUND`	600 s	loop must tick within this
`DEFAULT_MAX_CONCURRENT`	1	workers per runner
`DEFAULT_MAX_RSS_RECYCLE_MB`	2048	recycle the process above this
`RUNNER_SERVICE_RESTART_EXIT_CODE`	75	exit code asking for restart

Constant	Value	Purpose
DEFAULT_TASK_TIMEOUT	1800 s	wall clock per run
DEFAULT_GOVERNANCE_POLL_INTERVAL	5 s	kill and deadline checks
DEFAULT_TOKEN_SAMPLE_INTERVAL	30 s	token usage sampling
DEFAULT_TERM_GRACE_SECONDS	30 s	SIGTERM to SIGKILL
DEFAULT_SHUTDOWN_DRAIN_SECONDS	300 s	graceful shutdown wait
DEFAULT_MAX_RETRIES	3	execution retries per note
INTERRUPTED_CHILD_MAX_LAUNCHES	3	relaunches of one child

Constant	Value	Purpose
DEFAULT_REAPER_THRESHOLD	300 s	note considered stalled
DEFAULT_REAPER_INTERVAL	60 s	reaper cadence
DEFAULT_REAPER_LIVENESS_WINDOW	300 s	recent-activity window
DEFAULT_SOFT_AFFINITY_TIMEOUT	300 s	soft host pin ages out
DEFAULT_DISPATCH_STUCK_THRESHOLD	30 cycles	watchdog trips
DEFAULT_DISPATCH_STUCK_RESTART_COOLDOWN	900 s	between self-restarts
DEFAULT_DEGRADE_FAILURE_THRESHOLD	5	failed checks before degrade

Constant	Value	Purpose
DEFAULT_BRANCH_PUSH_RETRIES	3	branch push attempts
DEFAULT_BRANCH_PUSH_RETRY_BACKOFF_SECONDS	2 s	between push attempts
ACCEPTANCE_CHECK_TIMEOUT_SECONDS	120 s	acceptance probe
TUI_SMOKE_TIMEOUT_SECONDS	600 s	TUI smoke check
SESSION_CHECKPOINT_INTERVAL_SECONDS	300 s	session capture cadence
DEFAULT_WORKSPACE_HYGIENE_INTERVAL	3600 s	worktree cleanup pass
DEFAULT_WORKTREE_MIN_FREE_GB	5 GB	low-disk alert

Two related values live elsewhere. A runner counts as a live advertiser of a dispatch lane for 300 s after its last heartbeat (DEFAULT_RUNNER_HEARTBEAT_FRESH_SECONDS, dispatch_target_resolution.py). A rate-limited host plan cools down for 300 s and by default holds its work (DEFAULT_COOLDOWN_SECONDS, DEFAULT_RATE_LIMIT_BEHAVIOR = "hold", host_plans.py).

B.16. Planning Timers

Constant or field	Value	File
`PlanningExecutionConfig.lease_seconds`	300 s	`planning_execution.py`
`PlanningExecutionConfig.deadline_seconds`	1800 s	`planning_execution.py`
`PlanningExecutionConfig.retry_backoff_seconds`	30 s	`planning_execution.py`
`max_attempts` fallback in `fail_job`	3	`planning_jobs.py`

Constant	Value	File
`EXPECTED_PLANNING_CONTROLLER_INTERVAL_SECONDS`	10 s	`planning_jobs.py`
`PLANNING_JOB_STALL_AFTER_SECONDS`	300 s	`planning_jobs.py`
`PLANNING_LEASE_EXPIRY_GRACE_SECONDS`	30 s	`planning_jobs.py`
`DEFAULT_MAX_CONCURRENT` (fleet planners)	1	`planner_fleet.py`
`MIN_CHILD_TASKS`	3	`contracts/planned_root.py`
`PLAN_CONTRACT_VERSION`	`"5.0"`	`contracts/planned_root.py`

The runner carries the same three planning values as `planner_lease_seconds`, `planner_deadline_seconds` and `planner_retry_backoff_seconds` on `RunnerConfig`. Child outcome text is limited to 200 characters per check summary and 400 for the assertion (`MAX_CHECK_SUMMARY_CHARS`, `MAX_ASSERTION_CHARS`).

B.17. Refinery Timers

The first table gives code defaults. The second puts the code default beside the value in the reviewed controller document on `main`, `ops/yesod-dispatch/config/controller-reviewed-document.json`. That file is the reviewed mirror of the controller's configuration, not a capture of the live host.

Code default	Value	File
`gate_reconciliation_seconds`	3600 s	`refinery_controller.py`
`deployment_settle_seconds`	86,400 s	`refinery_controller.py`
`max_repairs_per_note`	2	`refinery_conflict_repair.py`
`STALL_GRACE_MINUTES`	15 min	`refinery_queue.py`
`DEFAULT_RUNNING_CAP` (pre-gate)	2	`refinery_pre_gate.py`
`DEFAULT_PRE_GATE_TIMEOUT_SECONDS`	1200 s	`refinery_pre_gate_runner.py`
`TARGET_GATE_RED_PERCENT`	10	`refinery_pre_gate_metrics.py`

The first two are fields of `LaneControllerConfig`; the third is a field of `ConflictRepairPolicy` and accepts 1 through 10.

Key	Code default	Reviewed document
`poll_interval_seconds`	5 s	5 s
`max_active_lanes`	2	2
`lease_seconds`	600 s	30,600 s
`renew_before_seconds`	180 s	15,300 s
`max_repairs_per_note`	2	1
`gate_reconciliation_seconds`	3600 s	7200 s (`yesod` lane)
`sefer_submit_timeout_seconds`	300 s	300 s
`sefer_observe_timeout_seconds`	30 s	30 s
`sefer_cancel_timeout_seconds`	60 s	60 s

The code defaults in this table are the fallbacks the configuration loader in `refinery_runtime.py` applies when a key is absent, except `max_repairs_per_note` (`ConflictRepairPolicy`). `gate_reconciliation_seconds` is set per refinery lane: the reviewed document gives 7200 s for the `yesod` lane, 1200 s for four of the other five lanes and 3600 s for the fifth.

B.18. Gate Limits

Constant	Value	File
`SEFER_MIN_SHARD_COUNT`	8	`refinery_gate_partition.py`
`SEFER_MAX_SHARD_COUNT`	16	`refinery_gate_partition.py`
`SEFER_PROTOCOL_VERSION`	2	`refinery_sefer_gate.py`
`DEFAULT_PARALLEL_CHUNK_COUNT`	200	`refinery_gate_chunks.py`
`MAX_JOBS` (gate spec)	128	`gate_spec/v2_model.py`
`ROLE_DEADLINE_SECONDS`	7200 s	gate forced command

`ROLE_DEADLINE_SECONDS` is written as `120 * 60` in `ops/sefer-gate/bin/yesod-sefer-gate-forced-command`, the worker-side script. The reviewed document sets `shard_count` 8 for the `yesod` lane, with `shards_per_candidate` 8 and `maximum_shards_per_candidate` 16.

B.19. Deployment Settle

Setting	Code default	Reviewed document
`settle_seconds`	86,400 s	1800 s (`yesod` lane)
`timeout_seconds`	120 s	600 s (`yesod` lane)

Both keys sit in a lane's `deployment` block. Their code defaults are `DEFAULT_SETTLE_SECONDS` and the `timeout_seconds` field of `CommandDeploymentConfig` in `refinery_deployment.py`; `settle_seconds` becomes `LaneControllerConfig.deployment_settle_seconds`. Only the `yesod` lane has a `deployment` block in the reviewed document. `STAGE_RETRY_BACKOFF_SECONDS` in `routine_deployment.py` is 5 s, then 15 s.

An attempt still in `deploying` when the settle time runs out ends `merged_not_deployed` with reason `deployment_settle_timeout` (`LaneController._merged_not_deployed`); Chapter 10 traces the path.

B.20. Replacement Timers

Constant or field	Value	File
`REPLACEMENT_PROTOCOL_VERSION`	2	`replacement_contract.py`
`lease_seconds` (authority)	900 s	`replacement_contract.py`
`successor_activation_seconds`	600 s	`replacement_contract.py`
`addresses_release_seconds`	60 s	`replacement_contract.py`
`UNMEASURED_SUCCESSOR_BRINGUP_SECONDS`	1800 s	`replacement_contract.py`
`UNMEASURED_ENROLLMENT_TTL_SECONDS`	1800 s	`replacement_contract.py`

The three lowercase rows are defaults of fields on `ReplacementSuccessorContract`. The two `UNMEASURED_*` values are placeholders: the driver has a typed refusal, `bringup_bounds_unmeasured`, for a contract that still carries them.

Constant or column	Value	File
`deadline_seconds` (intent)	14,400 s	migration `0117`
`claim_hold_seconds` (intent)	30 s	migration `0117`
`MAXIMUM_ROLLOUT_SECONDS`	7200 s	`replacement_cli.py`
`SWAP_CONCURRENCY`	1	`runner_replacement.py`
`DEFAULT_PREDECESSOR_STALE_SECONDS`	1200 s	`replacement_driver.py`
`MAX_PROOF_TIMEOUT_SECONDS`	300 s	`replacement_adopt.py`
`MAX_CADENCE_SECONDS`	90 s	`replacement_preflight.py`

The two intent columns are defaults on `replacement_intents`; the table allows 1 to 300 s for the claim hold and 1 to 86,400 s for the deadline. `MAXIMUM_ROLLOUT_SECONDS` is also defined, with the same value, in `runner_replacement.py`.

Unit or exit code	Value	Source
`yesod-replacement-driver.timer`	every 1 min	`ops/yesod-dispatch/systemd/`
`yesod-successor-activate.timer`	every 30 s	`ops/yesod-dispatch/systemd/`
`EXIT_ROLLED_BACK`	2	`replacement_cli.py`
`EXIT_NEEDS_OPERATOR`	3	`replacement_cli.py`
`EXIT_NONTERMINAL`	75	`replacement_cli.py`

B.21. Mail, Bulletin and Fleet

Constant	Value	File
`ROSTER_TTL_SECONDS`	1800 s	`mail_store.py`
`HOSTED_MAIL_SENDS_PER_HOUR`	60	`mail_store.py`
`BRIDGE_SJBIS_DEFAULT_INTERVAL`	30 s	`mail_cli.py`
`mail ask --deadline` default	`5m`	`mail_cli.py`

A roster row not refreshed within `ROSTER_TTL_SECONDS` is treated as stale (`prune_stale_agents`).

Constant	Value	File
`DEFAULT_MAX_POST_BYTES`	16,384	`bulletin_store.py`
`DEFAULT_MAX_COMMENT_BYTES`	4,096	`bulletin_store.py`
`DEFAULT_POSTS_PER_HOUR`	20	`bulletin_store.py`
`DEFAULT_COMMENTS_PER_HOUR`	60	`bulletin_store.py`

These are defaults; stored board settings can override them (`update_settings`, `bulletin_store.py`).

Constant	Value	File
`STOP_EXIT_CODE`	42	`fleet_cli.py`
`HARD_STOP_ESCALATE_SECONDS`	2 s	`fleet_cli.py`
`RUN_STUCK_THRESHOLD_SECONDS`	900 s	`factory_telemetry.py`
`HEARTBEAT_STALE_THRESHOLD_SECONDS`	120 s	`factory_telemetry.py`

B.22. Scratch Leases

Scratch limits are reviewed configuration, stated in `docs/policies/scratch-vms.md` on `main`. The policy says they are "configuration values, not fallback guesses in code", and no matching constants were found in `scratch_service.py`.

Setting	Reviewed default
Lease time limit	2 hours
Maximum requested from now	24 hours
Per-agent cap	2 concurrent leases
Fleet cap	12 concurrent leases

Source Notes

Section	Tag	Anchor
Note status, disposition	[code]	`note_lifecycle.py` (`IMPLEMENTATION_LIFECYCLE`, `USAGE_LIFECYCLE`, `CONTRACT`); `disposition.py` (`DISPOSITIONS`); `note_lifecycle_ops.py`; `db.py`
Planning jobs	[code]	`planning_jobs.py` (`ACTIVE_STATES`, `TERMINAL_STATES`, `fail_job`); `planning_execution.py`; `planning_publication.py`; `planner_fleet.py`
Runs, child outcomes	[code]	`run_ledger.py`; `child_outcomes.py`; `contracts/planned_root.py`
Attempts, gate executions, evidence	[code]	`refinery_state.py` (`AttemptState`, `GateExecutionState`, `ExternalEffectEvidence`); `refinery_pre_gate.py`
Admissions	[code]	`refinery_admission.py` (`AdmissionState`); migration `0045_refinery_cutover_ys_yes_x802.py`
Replacement phases, targets, journal, stop guards	[code]	`replacement_contract.py`; migrations `0117_replacement_promotion_state_ys_yes_mabj.py` and `0119_runner_replacement_retention_ys_yes_mabj.py`
Runner timers	[code]	`runner.py`; `dispatch_target_resolution.py`; `host_plans.py`

Section	Tag	Anchor
Refinery, gate, settle	[code]	`refinery_controller.py`; `refinery_runtime.py`; `refinery_deployment.py`; `refinery_conflict_repair.py`; `refinery_queue.py`; `refinery_pre_gate*.py`; `refinery_gate_partition.py`; `refinery_gate_chunks.py`; `refinery_sefer_gate.py`; `gate_spec/v2_model.py`; `routine_deployment.py`; `ops/sefer-gate/bin/yesod-sefer-gate-forced-command`
Reviewed controller values	[deploy]	`ops/yesod-dispatch/config/controller-reviewed-document.json` at `16606a42`; a reviewed mirror, not a live capture
Replacement timers	[code]	`replacement_contract.py`; `replacement_cli.py`; `replacement_driver.py`; `replacement_adopt.py`; `replacement_preflight.py`; `runner_replacement.py`; `ops/yesod-dispatch/systemd/*.timer`
Mail, bulletin, fleet	[code]	`mail_store.py`; `mail_cli.py`; `bulletin_store.py`; `fleet_cli.py`; `factory_telemetry.py`
Scratch leases	[deploy]	`docs/policies/scratch-vms.md` at `16606a42`
Swap status	[deploy]	agents' working notes, 2026-11-15 to 2026-11-19; see Chapter 11

Not verified:

- The values a live host is running. Every number here is a code default or a value in a reviewed file on `main`; none was read from a running controller or runner.
- The one-line meanings in the state tables are summaries. Exact entry and exit conditions are in the transition tables named beside each vocabulary (`_ATTEMPT_TRANSITIONS`, the triggers in migration `0117`) and in the chapters that own each mechanism.
- The "set by" column for dispositions, and the meaning of each stop-guard state, were not traced end to end.
- The list of planning `terminal_reason` codes is a sample found by search, not a complete enumeration; the codes are string literals, not an enum.
- Migration `0119` replaces the target-state trigger function (the table above reflects it). The other replacement constraint and trigger names were searched for in later migrations and found only in `0117` and `0119`; the `0119` contract declarations were not compared line by line.
- That the first lane in the reviewed document is the `yesod` lane is read from its repository path and Sefer gate provider; the document does not name a tool per lane.

C Roles and Authority

This appendix lists who and what acts in the factory, and what stops each of them from acting outside its job. It has four parts: the roles defined in code, the cast of agents that ran the factory in October and November 2026, the deterministic services, and the authority rules with the thing that enforces each one.

Two layers of "role" exist and they do not match. The first is a set of primers in the source, written between June and August. The second is the group of long-lived agent sessions that operated the factory in the autumn, most of which have no primer. Chapter 17 (The Agent Organization) tells that story. Module names below are relative to `yesod/src/yesod/` in `yesod-aicoe` at `origin/main` `16606a42` unless a directory is shown.

C.1. Primed Roles in Code

`yesod prime` (`prime.py`, `register()`) always prints `PRIME_GUIDE` and appends one guide for each flag given. There are ten role flags and two addenda. A primer is text that an agent reads at the start of a session. It is not a credential and nothing checks that a session primed for one role stays inside it.

Flag	Constant	Designed job
`--worker`	`WORKER_GUIDE`	implement one note in a worktree
`--planner`	`PLANNER_GUIDE`	turn a note into a bead tree, leave it unarmed
`--dispatcher`	`DISPATCHER_GUIDE`	choose dispatch lanes, arm and hold
`--refinery`	`REFINERY_GUIDE`	human-guided merge work; shows the live queue
`--mayor`	`MAYOR_GUIDE`	human interface, routing, arbitration
`--triage`	`TRIAGE_GUIDE`	incident evidence and a recovery plan
`--infra-monitor`	`INFRA_MONITOR_GUIDE`	fleet health
`--mad-scientist`	`MAD_SCIENTIST_GUIDE`	controlled improvement experiments

Flag	Constant	Designed job
`--ephemeral-refinery-supervisor`	(see below)	oversight of the retired merge daemon
`--overlord`	`OVERLORD_GUIDE`	keep role seats alive

The constant for the ninth flag is `EPHEMERAL_REFINERY_SUPERVISOR_GUIDE`. The worker guide is built from `_WORKER_PREAMBLE` plus the shared worker text in `worker_canon.py`, which is also rendered into `AGENTS.md`. The refinery guide appends a live section from `_merge_queue_section()`.

Addendum flag	Constant	Adds
`--use-mail`	`MAIL_USAGE_ADDENDUM`	how to use `yesod mail`
`--use-sjbis`	`HUMAN_CHANNEL_ADDENDUM`	when and how to ask the human

`BULLETIN_USAGE_ADDENDUM` is not a flag. It is concatenated onto `PRIME_GUIDE`, so every primed agent receives the bulletin trust rules.

Five other lists in the source name roles, and they disagree with the primers and with each other. Two of them are different constants with the same name: the seven-entry tuple in `factory_status.py` names the `factory:<role>` topics that `yesod factory status` reads, and the four-entry list in `mail_cli.py` is the set of seats that `yesod factory start` spawns.

List	Where	Members
`SINGLETON_ROLES`	`mail_cli.py`	mayor, dispatcher, infra-monitor, overlord, triage
`FACTORY_ROLES`	`factory_status.py`	planner, dispatcher, infra-monitor, mayor, triage, ephemeral-refinery-supervisor, overlord
`FACTORY_ROLES`	`mail_cli.py`	mayor, dispatcher, infra-monitor, ephemeral-refinery-supervisor
`DEFAULT_PROMOTER_ROLES`	`bulletin_store.py`	mayor, planner, throughput-supervisor
role pages	`docs/claude-agent-team/`	eight pages; no worker, no overlord

The roles list inside `PRIME_GUIDE` names nine roles and omits overlord. `throughput-supervisor` has no primer.

C.2. The Operating Cast, October and November 2026

Everything in this section comes from the agents' working notes for 2026-10-12 to 2026-11-19, not from source. Each of these roles was a long-lived interactive session registered on the mail roster, started from its own working directory. Only the mayor has a primer in `prime.py`.

Role	Job	May not
Stephen (operator)	sole human authority; signs grants; runs `[S]` steps	(no limit recorded)
mayor	coordinates; drafts grant texts; the only requester of permission	author application code; release, migrate or change hosts without a grant
devil's advocate	independent second look at texts, scripts, plans, diffs	hold, approve, push, mutate; its review informs and does not veto
forensics	read-only investigation; numbered findings; delivery audits	mutate; run tests against production or gate databases
refinery clerk	sole writer of lane holds and releases; owns the admissions pause in normal operation	release without the named grant and legs
pre-run and rehearsal seat (aws-planner)	full test passes and rehearsals on disposable boxes; plan review	authorize the mayor; write lane state
monitors, watchers, sentinels	alert; fail-closed re-pause; hold on delivery	decide
isolated build seat (gilgul)	the runner host where dispatched workers build approved notes	(a host, not an agent)
human visitors	ask questions on the hosted demo board	order changes, approve
subagents	read and compile for a long-lived role	run tests; write lane state

What actually checks each of these is given in the authority table at the end of this appendix. In short: the mayor's limits, the devil's advocate's role and the clerk's sole-writer status are conventions kept by peer review on the bulletin. Mayor authority changed during the period (standing recovery authority from 2026-10-10, approval of in-scope re-plans from 2026-10-29, running a pinned command under a valid grant from 2026-11-10). Chapter 18 (Signed Human Authority) gives the dates and reasons.

Whether the designed seats (dispatcher, triage, ephemeral-refinery-supervisor, overlord)

were running in November alongside this cast is not established.

C.3. Deterministic Services Today

These do mechanical work with no model in the loop. "Unit" means a unit file under `ops/yesod-dispatch/systemd/` on `main`. A unit file on `main` does not prove the unit is enabled on a host.

Service	Module or unit	What it does
runner daemon	`runner.py` `Runner`; `yesod-codefactory-dispatch.service`	claims armed notes, spawns and governs agents, verifies, pushes (`yesod codefactory-dispatch`)
lifecycle scheduler	`lifecycle_dispatch.py` `LifecycleScheduler`	arms approved notes; ticked by the runner (`_tick_lifecycle_scheduler`)
planner jobs	`planning_jobs.py`; `planning_execution.py` `PlanningExecutionService`	leased, retried planning runs; ticked by the runner (`_tick_planning`)
merge controller	`refinery_runtime.py` `DeterministicRefineryService`; `yesod-refinery-controller.service`	steps each attempt: prepare, gate, push, deploy step (`yesod refinery controller`)
gate providers	`refinery_gate_provider.py` `GateProviderName`; `refinery_gate_broker.py`	`local`, `microvm`, `sefer`, `container`; the Yesod lane uses `SeferGateProvider`
replacement driver	`replacement_driver.py` `ReplacementDriver`; `yesod-replacement-driver.timer`	one pass a minute over a replacement intent (`yesod replacement drive`); ships disabled
successor activation unit	`successor_activation.py`; `yesod-successor-activate.service` and `.timer`	on the successor, brings services up after the database proves the handover; retries every 30 s
deployment observer	`deployment_observer.py`; `yesod-deployment-observer.timer`	records the three-surface deployment observation every minute
exporter	`factory_telemetry.py` `FactoryTelemetryExporter`; `yesod-factory-exporter.service`	reads committed factory events and sends OpenTelemetry (`yesod factory-exporter run`)

Service	Module or unit	What it does
seat keeper	`ops/pompom/scripts/factory-seat-keeper.sh`	interim job written to respawn missing role seats every 10 minutes; see the note below
sjbis bridge	`mail_cli.py` (`yesod mail bridge-sjbis`); `yesod-bridge-sjbis.timer`	every 30 s, passes mail for the human address to the external `sjbis` tool

The seat keeper script reads a `missing_roles` field from the output of `yesod factory status`. No code under `yesod/src/yesod` on `main` emits that field for the status report any longer, so the script as it stands on `main` finds nothing to respawn.

Three more unit files on `main` belong to older or supporting paths. `yesod-release-promotion.service` runs the release promotion worker, whose in-place body is now the legacy mode (Chapter 10, Merged Is Not Deployed). `yesod-gate-fleet-poller.service` is the MicroVM fleet poller that `ops/yesod-dispatch/README.md` calls legacy. `yesod-replacement-host-local.socket` and `yesod-successor-enroll.path` serve the single-use enrollment of a successor (Chapter 11, Deploy Is a Host Swap).

C.4. Authority Rules

Each rule names what enforces it. A symbol name means a check in code on `main`. "Convention" means that nothing in the factory repository stops a violation; the rule is kept by primers, by the working rules of the cast, and by peers reading the bulletin. Nothing in the repository verifies a human signature. Grants are signed and verified with the external `ta` tool.

Rules with a check in code:

Rule	Enforced by
A bulletin author is a roster address or the interactive operator	`resolve_author` in `bulletin_identity.py`
Every bulletin read is labelled untrusted peer content	`BULLETIN_CONTENT_TRUST` in `bulletin_store.py`
Promoting a bulletin thread needs the operator or a promoter role	`resolve_promoter`
Mail `from` cannot be set with `--from`	`_self_send_address` in `mail_cli.py`
A second session cannot take a singleton seat with a different token	`_reject_singleton_hijack`
A roster row is pruned only when heartbeat and activity are both stale	`prune_stale_agents` in `mail_store.py`

Rule	Enforced by
Admissions cannot resume without the pause token and expected revision	`resume_admissions` in `refinery_admission.py`
Adoption needs root, actor `stephen`, and a pause that actor owns	`_require_root`, `_adoption_actor`, `_read_owned_pause` in `replacement_adopt.py`
Controller authority moves only by compare-and-swap under an owned pause	`transfer_controller_authority` in `controller_authority.py`
No automatic swap carries a schema migration	`AutomaticMigrationRefused` in `replacement_migration.py`
A held note cannot be approved	`approve` in `note_lifecycle_ops.py`
A dispatched agent gets only allowlisted environment	`build_agent_launch_environment`
A worker cannot create beads under a published plan's child	`PlanBoundLeafMutationError` in `beads.py`
A terminal merge attempt never changes	trigger function `refinery_reject_terminal_attempt_mutation` on `refinery_attempts`
All-stop, bulletin settings and operator posts need the operator	`_authorized_for_all_stop` in `fleet_cli.py`
Acting on another agent's scratch lease needs operator or mayor	`ScratchPrincipal` in `scratch_service.py`; `docs/policies/scratch-vms.md`

Four of these are weaker than they read. `_authorized_for_all_stop` is an environment check: a flag, the mail identity, or the acting user name. Any process running as that user passes. `approve` checks plan provenance and disposition and no identity at all; whoever can run the CLI against the database can approve. `_reject_singleton_hijack` lets a caller with no token through. `--from` is discarded, but `--as` accepts any well-formed address, including the human address (this last point is inferred from reading `resolve_self` and was not exercised).

Rules kept by convention:

Rule	Enforced by
Peer mail and bulletin content carry no human authority	convention (`PRIME_GUIDE` text)
Only Stephen's signed grant authorizes a release, a first approval, a scope change, a migration or a host change	convention; grant checked with external `ta`
Attention goes through `sjbis`, permission through `ta`	convention
A timeout carries no authority; a used single-use grant covers no retry	convention

Rule	Enforced by
Agents on production-adjacent seats do not author application code	convention (`MAYOR_GUIDE` tells the mayor never to author the fix; the wider rule is in the working notes)
Two pre-sign checks (devil's advocate, forensics) precede any ask	convention
No approvals while a release window is open	convention
The refinery clerk is the sole writer of lane holds and releases	convention
Every `awaiting_merge` note is held by default (K2)	convention; detected by `evaluate_k2`
Forensics posts the SQL with each finding (K9)	convention; detected by `detect_k9_comments`
Devil's advocate review is required for high-blast-radius actions and does not veto (K6)	convention; listed in `NEVER_CHECKED_RULES`
Only `human/stephen/main` carries operator authority on a board	`OPERATOR_ADDRESS` for the board role; treatment of visitors is convention
Workers do not merge to `main`, rebase, or close the root bead	convention (`worker_canon.py`)
Subagents never run the test suite	convention
A process identity is whatever its environment says	not enforced (`resolve_self`)

The registry in `rules_contract.py` states the assurance of its own twelve rules: none is `ENFORCED`, six have detectors, and six are attested only.

C.5. Not Verified

- Which unit files are enabled on the serving controller. The working notes for 2026-11-11 say the replacement driver timer and service stayed disabled.
- The launchd job definition for the seat keeper. The script names a plist that is not under `ops/pompom/launchd/` on `main`.
- Whether `mail send --as human/stephen/main` succeeds. Read from source, not run.
- Whether every agent on the control machine satisfies `_authorized_for_all_stop`. This is inferred from the user-name fallback and was not checked on the host.
- Which designed seats were running in November.
- The internals of `ta` and `sjbis`. Both are outside the repository.

D Evidence and Source Map

This appendix says what the book's claims rest on. It gives the source snapshot, the tiers of evidence, a table of source anchors for each chapter, and a list of the facts that rest only on the agents' working notes.

D.1. The Snapshot

Item	Value
Source repository	`yesod-aicoe`
Commit read	`origin/main` at `16606a42`, commit date 2026-11-17
Verification date	2026-11-19
Prior edition baseline	`4d658cc6` (2026-08-16), last edited 2026-08-20

Every path in this appendix was checked to exist in the tree of `16606a42`. Source was read from the commit, never from a working checkout. No test suite was run and no production host or database was touched to write this edition. Line numbers are not cited because they move; the anchors are file paths and symbol names. Commit dates are committer dates in UTC. The snapshot commit is itself a candidate merge commit, and its timestamp (2026-11-17 19:16Z) is derived from its source commit; it is not the time the merge landed, which the working notes place on 2026-11-18.

A bare module name such as `runner.py` means `yesod/src/yesod/runner.py`. Paths that begin with `ops/`, `docs/`, `config/` or `decisions/` are relative to the repository root. Runbooks are under `docs/runbooks/`.

D.2. Evidence Tiers

The book uses four kinds of evidence. They are not equal.

Tier	What it is	How it appears in the text
code	behavior read in source at the snapshot	a symbol, table, state or verb in `code font`
deploy	a dated fact about the live system	a date plus a runbook file or commit
notes	a fact recorded only in the agents' working notes	"the agents' working notes record …" with a date
inferred	a reading not traced end to end	the sentence says "inferred" or "not traced"

A commit proves that code exists on `main`. It does not prove that the code is installed on a host, enabled, or exercised. A runbook on `main` that records an event (for example the controller handoff of 2026-09-16) is deployment evidence for that date. A runbook that describes a procedure is evidence of design only.

The agents' working notes are the files that the coordinating agents kept for themselves while they operated the factory: a memory directory of short dated notes and the mayor's state and handoff files. An agent wrote each one at the time, for its own later use. They were not reviewed, they are not in the factory repository, and some record a belief that a later note corrects. The book uses them because from October onward they are often the only record of why something happened: non-merge commit bodies total 238,589 bytes for September and 37,250 bytes from October 1 onward (`git log --no-merges --format=%b | wc -c` on `origin/main`, by committer date in UTC). Where a commit on `main` corroborates a note, the commit is cited. Where none does, the text says the fact is recorded in the notes. The bulletin board, which is the cast's primary record, was not read for this edition.

D.3. Source Anchors by Chapter

D.3.1. 1. Follow the Change

Claim area	Anchors
Note statuses in order	`note_lifecycle.py` (`IMPLEMENTATION_LIFECYCLE`)
Planning, approval, arming	`planning_jobs.py`; `note_lifecycle_ops.py` (`approve`); `lifecycle_dispatch.py` (`LifecycleScheduler`)
Claim and child execution	`runner.py` (`_claim_note`); `root_dispatch_reconciliation.py`
Attempt, candidate, push	`refinery_state.py` (`AttemptState`); `refinery_git.py`

Claim area	Anchors
Gate union	`refinery_sefer_gate.py`; `refinery_gate_partition.py`
`deploying`, `merged_not_deployed`	`refinery_controller.py` (`_run_deployment`)
Two PRs merged outside the gate, 2026-09-14	commit `0373aa5b4`

D.3.2. 2. Judgment and Mechanism

Claim area	Anchors
Ten role flags, two addenda	`prime.py` (`register`, `PRIME_GUIDE`)
Identity and roster	`mail_cli.py` (`resolve_self`); `mail_store.py` (`prune_stale_agents`)
What code enforces	`fleet_cli.py` (`_authorized_for_all_stop`); `replacement_adopt.py` (`REQUIRED_ACTOR`)
Integration plane	`refinery_runtime.py`; `refinery_gate_provider.py`; `replacement_driver.py`
Receipts and journals	`replacement_receipts.py`; migration `0117_replacement_promotion_state_ys_yes_mabj.py`

D.3.3. 3. The Two Currencies

Claim area	Anchors
Note ids and prefixes	`db.py` (`generate_note_id`, `tool_prefix`)
Dispositions	`disposition.py` (`DISPOSITIONS`, `effective_disposition`)
Audited writes	`note_status_writer.py` (`write_status`, `write_disposition`)
Note dependencies	`lifecycle_dispatch.py` (`_prerequisites`); `note_dependency_policy.py`
Merged but undeployed prerequisites	`refinery_intake.py` (`_project_merged_not_deployed`); `note_settlement_read.py`
Beads routing and remote Dolt	`beads.py` (`run_bd`, `resolve_dolt_connection`)
Plan-bound leaves	`beads.py` (`PlanBoundLeafMutationError`)

D.3.4. 4. The Registry Beneath the Factory

Claim area	Anchors
Tools, notes, tags	`tool.py`; `notes.py`; `note_tags.py`
`yesod ask` ranking and budget	`retrieval.py` (`retrieve_catalog`); `embeddings.py`; `ask.py`
Graph evidence	`graph_retrieval.py` (`GraphEvidenceClient`)
Processes and molecules	`processes.py`
Onboarding and fleet policy	`onboarding.py`; `onboarding_model.py`; `config/onboarding-fleet.yaml`
Forge-neutral contract	`forge.py`; `forge_gitlab.py`
Worker canon	`worker_canon.py` (`render_canon`); `AGENTS.md`

D.3.5. 5. Capture, Plan, Deliver

Claim area	Anchors
Statuses and transitions	`note_lifecycle.py` (`CONTRACT`, `LifecycleTransition`)
Planner jobs, leases, generations	`planning_jobs.py` (`acquire_lease`, `fail_job`)
Plan contract and open questions	`planning_contract.py` (`validate_plan_artifact`); `contracts/planned_root.py`
Revision binding	`planning_publication.py`; `planning_input.py`
Approval	`note_lifecycle_ops.py` (`approve`)
Automatic arming	`lifecycle_dispatch.py` (`claim_arm_sql`); table `lifecycle_dispatch_arms`
Claimability	`claimable.py`; `why_not_running.py` (`collect_evidence`)
Admissions pause scope	`refinery_admission.py` (`require_admissions_open`)
Inspecting a stuck plan	`planning-job-inspection.md`

D.3.6. 6. The Runner

Claim area	Anchors
Loop, claim, worker pool	`runner.py` (`Runner`, `_run_poll_cycle`, `_claim_note`)
Planner and lifecycle ticks	`runner.py` (`_tick_planning`, `_tick_lifecycle_scheduler`)
Governance and timers	`runner.py` (`_wait_with_governance`, `RunnerConfig`)
Memory recycle	`runner_memory.py`
Run ledger	`run_ledger.py` (`insert_starting_run`, `finalize_run`)
Heartbeat	`dispatch_agent_log.py`; table `dispatch_log`
Watchdog incident, 2026-09-10	`runner.py` (`_tick_dispatch_watchdog`); commit `6330a83aa`
Claim lock change	commit `10090d261`

D.3.7. 7. Governed Agent Execution

Claim area	Anchors
Worker rules	`worker_canon.py` (`SECTIONS`)
Launch environment	`agent_launch_environment.py`; `docs/security/agent-launch-environment.md`
Child outcome protocol	`child_outcomes.py` (`CHILD_OUTCOME_HEARTBEAT_PREFIX`)
Acceptance evidence	`acceptance_evidence.py` (`classify_child_acceptance`)
Child close reasons	`root_dispatch_reconciliation.py` (`child_close_reason`)
Execution providers	`execution_provider.py`; `execution_broker.py`; `lambda_microvm_execution.py`
Recovery verbs	`retained_child_recovery.py`; `retained-child-recovery.md`

D.3.8. 8. The Merge Controller

Claim area	Anchors
Attempt states and transitions	`refinery_state.py` (`AttemptState`, `authorize_attempt_transition`)
Claim, lease, lane slot	`refinery_control.py` (`reconcile_terminal_lane_slots`)
The tick	`refinery_scheduler.py`; `refinery_controller.py` (`LaneController`)
Candidate commit and push	`refinery_git.py` (`candidate_ref_for`)
Intake	`refinery_intake.py`; `refinery_queue.py`
Leadership fence	`controller_authority.py`; migration `0118_controller_leadership_fence_ys_yes_mabj.py`
Admissions pause and token	`refinery_admission.py`; `docs/security/admissions-pause-token-surfaces.md`
Conflict repair	`refinery_conflict_repair.py`; `refinery_repair_lanes.py`
Pre-gate	`refinery_pre_gate.py` (`PreGateMode`)
Merge-turn theft, 2026-10-12	commit `779673ec0`

D.3.9. 9. The Deterministic Gate

Claim area	Anchors
Providers and contract	`refinery_gate_provider.py` (`GateProviderName`); `refinery_gate_broker.py`
Which lane uses which	`docs/refinery-fleet-gates.md`; `config/refinery-lanes.json`
Why the gate came home	commit `1964730c3`; `decisions/009-sefer-authoritative-refinery-gate.md`
Union and coverage proof	`refinery_sefer_gate.py`; `refinery_gate_partition.py` (`CoverageProof`)
Candidate carrier	`refinery_gate_state.py`
Forced commands	`ops/sefer-gate/bin/yesod-sefer-gate-forced-command`
Credentials on tmpfs	`sefer_gate_credentials.py`; `refinery_sefer_credentials.py`
Fleet preflight	`ops/yesod-dispatch/scripts/preflight-sefer-gate-fleet.py`

Claim area	Anchors
Rollout record	`sefer-refinery-gate-rollout.md`
Red runs counted green, fixed 2026-09-20	commit `cfb66cf03`

D.3.10. 10. Merged Is Not Deployed

Claim area	Anchors
Surfaces that drift	`release_skew.py` (`SurfaceObservation`)
August promotion transaction	`release_promotion_store.py`; `release_promotion_worker.py`
Routine deployment	`routine_deployment.py`; `routine_fleet_rollout.py`
`merged_not_deployed` state	migration `0092_refinery_merged_not_deployed_ys_yes_j0hc.py`
Settle bounds	`ops/yesod-dispatch/config/controller-reviewed-document.json`
Deployment observation	`deployment_observer.py`; `deployment_surfaces.py`
Sealed-tree guard	`controller_pytest_guard.py`; commit `2237c49e5`
Schema changes opt-in	`db.py` (`_migrate_allowed`); `baseline-ddl-replay.md`
Hand-run records, September	`controller-g1-handoff-2026-09-16.md`; `fleet-release-244a8ab1-2026-09-17.md`

D.3.11. 11. Deploy Is a Host Swap

Claim area	Anchors
Phases and target states	`replacement_contract.py` (`ReplacementPhase`)
Durable state	`replacement_store.py`; migration `0117_replacement_promotion_state_ys_yes_mabj.py`
Trigger	`routine_deployment.py` (`apply_candidate`); `replacement_cli.py`
Driver	`replacement_driver.py` (`ReplacementDriver`); `replacement_driver_condition.py`

Claim area	Anchors
Authority compare-and-swap	`controller_authority.py` (`transfer_controller_authority`)
Own pause, quiet gap	`replacement_guard.py` (`acquire_replacement_stop_guard`)
No automatic migration	`replacement_migration.py` (`AutomaticMigrationRefused`)
Enrollment	`replacement_enrollment.py`; `ops/yesod-birth/successor-enroll-stage0.py`
Handover, rollback, boundary	`replacement_handover.py` (`ControllerReplacementHandover`)
Activation	`successor_activation.py`
Adoption and preflight	`replacement_adopt.py`; `replacement_preflight.py`
Operator procedure	`replacement-driver.md`; `routine-deployment.md`

D.3.12. 12. Born, Not Repaired

Claim area	Anchors
Birth sequence and rule	`ops/yesod-birth/README.md`; `ops/yesod-birth/birth-common.sh`
Runner birth	`ops/yesod-birth/birth-runner.sh`; `runner_vm_bootstrap.py`
Gate birth	`ops/yesod-birth/birth-gate.sh`; `ops/yesod-birth/gate-profiles.json`
Controller birth	`ops/yesod-birth/birth-controller.py`; `controller-birth.md`
Generated gate inventory	`ops/yesod-birth/generate-gate-fleet-inventory.py`
Fleet sets	`ops/yesod-birth/manage-gate-fleet-set.py`
Host keys after reboot	`sefer-ssh-host-key-refresh.md`
Fleet control and secrets	`fleet_cli.py`; `fleet_crypto.py`; `secrets/`
Backups and blobs	`postgres_backup.py`; `blob_store.py`
Birth measurement	`ops/yesod-birth/measure-replacement-birth.py`

D.3.13. 13. Seeing the Whole System

Claim area	Anchors
Event table and envelope	`factory_events.py`; migration `0059_factory_events_ys_yes_rklm.py`
Event types	`factory_event_taxonomy/`
Exporter	`factory_telemetry.py` (`FactoryTelemetryExporter`)
Side streams	`gate_test_otel.py`; `runner_log_telemetry.py`; `ops/observability/render-log-collector.py`
Note-less attempt failures	`refinery_failure_journal.py`; commit `d587d4433`
Health projection	`factory_stats.py` (`collect_factory_stats`)
Per-note read models	`factory_observability.py` (`explain_note`); `live_viz_api.py`
Backend status	`docs/factory-observability-runbook.md`
Compliance rules	`rules_contract.py`; `rules_readers.py`

D.3.14. 14. Failure Is a State Machine

Claim area	Anchors
Execution failure	`reconciliation.py`; `failure_analysis.py`; `run_ledger.py`
Planning failure	`planning_jobs.py`; `planning_publication.py`
Integration failure	`refinery_state.py` (`TERMINAL_ATTEMPT_STATES`); `refinery_conflict_repair.py`
Gate failure	`refinery_state.py` (`GateExecutionState`)
Deployment and replacement	`replacement_contract.py` (`TerminalDisposition`); `release_skew.py`
Host failure runbooks	`sefer-gate-credential-lifecycle.md`; `sefer-gate-headroom-advisory.md`
Coordination	agents' working notes (no state in the database)

D.3.15. 15. The Reliability Contracts

Claim area	Anchors
Durable completion	`root_dispatch_reconciliation.py`; `completed_plan_reconciliation.py`
Its blind spot	`acceptance_evidence.py` (`acceptance_not_executed`, `db_env_unset`)
One writer by lease generation	`controller_authority.py` (`ControllerAuthority`)
Stop intake, prove who stopped it	`refinery_admission.py`; `replacement_guard.py`
Schema changes opt-in and ordered	`db.py` (`_migrate_allowed`); `lane_ordinals.py`
An unknown is never a zero	`release_skew.py`; commit `7bb582e12`
Rehearse a never-run path	agents' working notes, mid-November 2026
Wall-clock limit, enforced only by the owning runner	`runner.py` (`_wait_with_governance`); `run_ledger.py` (`request_kill`, called only from the CLI)

D.3.16. 16. Cost-Aware Dispatch

Claim area	Anchors
Dispatch lanes	`dispatcher/runners.py` (`RunnerRegistry`)
How arming picks a lane	`dispatch_target_resolution.py` (`require_live_advertiser`)
Cost capture	`dispatcher/usage.py`; `cost_accounting.py`; `feature_cost/`
Balance and parking	`refinery_repair_lanes.py`; `credit_exhaustion.py`; `host_plans.py`
Cloud gate spend cap	`refinery_cloud_gate_budget.py`
Gateway	`inference_gateway/`
Complexity estimate, unwired	`estimate_complexity.py`

D.3.17. 17. The Agent Organization

Claim area	Anchors
Primers	`prime.py`
Mail and roster	`mail_cli.py`; `mail_store.py`

Claim area	Anchors
Seats and liveness	`agent_cli.py`; `factory_status.py`; `ops/pompom/scripts/factory-seat-keeper.sh`
Bulletin authors and trust	`bulletin_identity.py` (`resolve_author`); `bulletin_store.py`
Boards and roles	`bulletin_board.py` (`board_role`)
Rules K1 to K12	`rules_contract.py` (`RULES`, `NEVER_CHECKED_RULES`)
Scratch machines	`scratch_service.py`; `docs/policies/scratch-vms.md`
The operating cast	agents' working notes, 2026-10-12 to 2026-11-19

D.3.18. 18. Signed Human Authority

Claim area	Anchors
Operator check	`fleet_cli.py` (`_authorized_for_all_stop`)
Approval has no identity check	`note_lifecycle_ops.py` (`approve`)
Pause token	`refinery_admission.py` (`pause_admissions`, `resume_admissions`)
Adoption binding	`replacement_adopt.py` (`_read_owned_pause`)
Human channel	`mail_cli.py` (`HUMAN_ADDRESS`, `resolve_human_channel`)
Same-window grant conditions	`replacement-driver.md`
Mail design intent	`docs/plans/2026-06-18-yesod-mail-protocol-design.md`
Grants, windows, legs, episodes	agents' working notes, 2026-10-24 to 2026-11-19

D.3.19. 19. What Failed, August to November

Claim area	Anchors
Commit counts by month	`git rev-list --count --since=2026-08-20T00:00:00Z origin/main`; months by committer date in UTC
Late August incidents	commits `c36105967`, `5751aa4eb`, `27dc12e01`

Claim area	Anchors
September incidents	commits `0373aa5b4`, `b014de872`, `2237c49e5`, `7a3e0a3f3`
Gate fleet soak	`soak-9114-2026-09-25.md`
October incidents	commits `779673ec0`, `9937e15f1`, `34e1d4085`, `c3451be82`
November code responses	commits `ad55cf61d`, `bcedcc1cb`, `5451304da`
Hours down, kill counts, findings	agents' working notes, dated in the text

D.3.20. 20. Patterns Worth Reusing

Claim area	Anchors
Replace hosts	`replacement_contract.py`; `ops/yesod-birth/README.md`
Merged but not running as a state	`refinery_state.py` (`AttemptState`)
Own your pause, prove quiet	`replacement_guard.py`
Point of no return in writing	`replacement-driver.md`
Dry run reports every failure	`replacement_preflight.py`
Registry states its assurance	`rules_contract.py` (`Assurance`)
Dormant by default	`hosted_config.py`; `hosted_server.py`

D.3.21. 21. The Road Ahead

Claim area	Anchors
Hosted API, dormant	`hosted_server.py`; `hosted_config.py`; `docs/hosted/README.md`
Product milestones	`docs/hosted/v1-release-gates.md`
AWS decisions	`decisions/008-aws-hosted-production-factory-re-anchor.md`; `decisions/010-aws-hosted-cdk-experiment-profile.md`
No live install	`aws-hosted-install-upgrade-recovery.md`; `aws-fargate-burst-pool.md`
Infrastructure code	`infra/aws_hosted/`; `ops/aws-hosted/`
Fleet and lanes	`config/onboarding-fleet.yaml`; `config/refinery-lanes.json`

Claim area	Anchors
Goal order, factories plan	agents' working notes, 2026-10-27 and 2026-11-15

D.4. What Rests on Working Notes Only

The following facts appear in the book and have no anchor on `main` except, in some cases, the commit that fixed the problem afterward. They are what an agent recorded at the time.

Controller generations and the swap campaign

- The dates of every hand-run controller host generation after 2026-09-16: g4 serving from 2026-10-26 00:24Z, g5 from 2026-11-03 00:59Z, and the three sittings of the cutover between 2026-10-31 and 2026-11-03. The handoff of 2026-09-16 has a runbook on `main`; the later ones do not.
- The three hand-submitted production replacement attempts of mid-November 2026, told in general terms in Chapter 11.
- The findings of the evening of 2026-11-19 (#130 to #136) and the state of the hardening patch, from the mayor's board notes.
- Adoption of the serving controller into the replacement lineage on 2026-11-10 at 23:20Z, and the four refusals before it.
- The green dry-run preflight on 2026-11-12, and the scratch-machine rehearsal that found further blockers after the third attempt.
- The state of the hardening work called H1: nineteen child tasks, delivered and in rework on 2026-11-19, not on `main`.
- That the replacement driver timer stayed disabled on the serving controller as of 2026-11-11.

Incident numbers

- 2026-09-10 watchdog: 36 fires, 17 of 17 drain timeouts, 979 agent-minutes lost against 2,346 productive.
- 2026-09-09: the list of eight silent failures and the four-hour lane stall.
- 2026-10-07 and 10-08: about 19 hours with the fleet down.
- 2026-10-13: 15 of 15 merges ending `merged_not_deployed` in 24 hours; 51 parked by 2026-10-14; 139 carried by 2026-11-03.
- 2026-10-29: fixture rows written into a production table by a builder agent (FINDING #58, which the notes date 2026-11-06).
- 2026-10-22: a peer blocked about nine and a half hours on a permission prompt.
- 2026-11-08: the planner VM stopped for 15 hours 45 minutes.
- 2026-11-09: an attempt minted 20 seconds before the hold.
- 2026-11-10: the 300-second cue missed 62 minutes after the window opened.
- 2026-11-19: all nineteen children closed with acceptance not executed.
- The 2026-09-27 host reset, its cause and the hardware changes that followed.

All governance practice

- The existence and duties of the devil's advocate, forensics, the refinery clerk, the pre-run seat, watchers and sentinels.
- Everything about grants: how `ta` is used, exit meanings, single-use grants, runbook-scope and standing grants, the ten-step request procedure, pre-sign legs, and the 10-minute cap on a blocking wait.
- Every dated rule and the mistake behind it (Chapter 18, episodes of 2026-09-03 to 2026-11-19), and every numbered finding. Finding numbers seen in the notes run from #37 to #130.
- The traced release window of 2026-11-11.
- Stephen's quoted statements and the dates of his decisions, including the rule of 2026-10-13 that deployment is a host swap and the goal order G1 to G5.
- The move of coordination from mail to the bulletin on 2026-10-12, and decision markers from 2026-10-13.

Deployment facts recorded in notes

- The current topology (which guests run on which host) as of 2026-10-22 to 2026-11-10.
- The three-node cluster of 2026-10-22 and the host memory upgrade of early October (one set of notes gives 2026-10-08, another 2026-10-09).
- That no AWS-hosted installation has run. Runbook status lines on `main` say the same (`live_install_status=not_run`); the notes do not contradict them.

D.5. Not Verified

- The individual causes and dates of the three production replacement attempts, which this edition leaves out.
- When the snapshot merge was pushed. The notes file the window under 2026-11-18; the commit's own timestamp is derived and is not a push time.
- Cutover dates for controller host generations g2 and g3.
- Whether the off-site handover rehearsal reached `succeeded`.
- The content of H1 and whether the pending migration precedes the fourth attempt.
- Production values of the replacement binding and whether the live controller document carries the deployment block.
- Transition order inside long modules (`runner.py`, `refinery_controller.py`, `replacement_driver.py`). State lists come from enum definitions; not every transition was traced.
- The bulletin board itself, the charter for numbering findings, and the text of the standing grant.
- Whether anything AWS-hosted has run live. `main` says no; confirm with the owner.

E Timeline, August to November 2026

This table places the events the book describes in date order, from the last edit of the previous edition (2026-08-20) to the verification date of this one (2026-11-19). Each row names its source.

The source column uses three tags. **code** means a commit on `origin/main` of `yesod-aicoe`, given by short hash; the date is the commit's committer date in UTC. **deploy** means a dated record of the live system in a runbook file on `main` (under `docs/runbooks/`). **notes** means the fact is recorded only in the agents' working notes and was not verified in source (Appendix: Evidence and Source Map). Where a row carries both "notes" and a commit, the event is from the notes and the commit is the code change that followed; where that commit carries a later date than the row, the row says so. A runbook that is a plan or a sentinel is evidence of what was planned, not of what completed, and those rows say so too. Times are UTC.

Read the table with one fact in mind. As of 2026-11-19 the first fully automatic, merge-triggered controller swap has not happened in production. Every controller change in this table was run by hand by an operator.

E.1. August

Date	Event	Source
08-20	Previous edition last edited; `main` holds 3,239 commits dated before 00:00 that day and 3,286 by the end of it	code: `git log --format=%ct`, counted by committer date
08-21	Screening model pinned to a path the API answers with 404; every screen fails closed as "unavailable"; a self-test is added	code: `c36105967`

Date	Event	Source
08-25	Hosted work is re-anchored on `main`; the side integration branch is retired as a target (the decision record is dated 08-24)	code: `3580e576b`; `decisions/008`
08-26	A second autonomous mayor session runs under the mayor's address (not verified; two accounts differ)	notes
08-28	The Sefer fleet becomes the pinned primary gate runtime in code; the note behind the commit records a candidate green on pristine CPython that went red in the MicroVM gate	code: `1964730c3`

E.2. September

Date	Event	Source
09-01	Last serial full-suite gate run ("Stage-0 R2")	deploy: `sefer-refinery-gate-rollout.md`
09-02	The parallel union lands on `main`: eight shards plus a safety island (the day it became the Yesod lane's authority is not recorded)	code: `2e77ba6b9`
09-02	Test units run against the production database host and leave 64 stray schemas	notes
09-06	Structured child outcomes, orphaned crash intents, planning job generations	code: `e17b48214`, `6f64d1d26`, `81ac0e880`
09-09	"Eight silent failures in one day", including a four-hour stall of the merge lane; lane slot repair committed 09-10	notes; code: `5a465dd9c`
09-09	The fleet birth recipe is vendored into the repository	code: `5f8582b8a`

Date	Event	Source
09-09	The bulletin board lands with verified authors	code: `de5355053`
09-10	The runner watchdog that was killing in-flight agents is suppressed while busy	code: `6330a83aa`; counts from notes
09-10	Gate database credentials are made persistent; the notes say a reboot had destroyed them	code: `a69954af0`; cause from notes
09-14	Two pull requests merged outside the gate leave `main` red; repaired	code: `0373aa5b4`
09-16	VM175 replaces the original controller VM, by hand; a normal gate under it passes at 20:27:01Z	deploy: `controller-g1-handoff-2026-09-16.md`
09-16	The Yesod lane switches to the dg5 gate fleet only; the nine old gate guests are shut down at 20:46:14Z and kept cold	code: `2584d2844`; deploy: `g1-cold-recovery-2026-09-16.md`
09-17	Fleet release `244a8ab1`, by hand, inside one pause window	deploy: `fleet-release-244a8ab1-2026-09-17.md`
09-18	A 4,000-byte truncation holds the lane in `deploying`; the state is bounded	code: `b014de872`
09-18	The plan of record for a combined controller window adds the routine deployment block to the live controller document	deploy (plan): `combined-controller-window-2026-09-18.md`
09-19	A sentinel document is written for a window to re-activate routine deployment for runner seats; no file on `main` records the first completed routine apply	deploy (plan): `routine-activation-sentinel-2026-09-19.md`
09-19	A test run writes into a sealed release tree and blocks promotion; guard committed 09-20	code: `2237c49e5`

Date	Event	Source
09-20	Three report readers that counted red runs as green are fixed	code: `cfb66cf03`
09-21	`yesod release-skew` added after three surfaces sat 0, 35 and 529 commits behind `main`	code: `7bb582e12`
09-21	An edit to the live controller document parks every attempt in `deploying`; fix committed 09-22	code: `7e8b9def6`
09-22	Stephen: “cattle, not pets”; rebirth runners instead of changing them in place	notes
09-25	48-hour soak of the new gate fleet begins at 06:29Z (closes 09-27 with 1 of 3 required green merges)	deploy: `soak-9114-2026-09-25.md`
09-27	The gate host hard-resets at 17:52Z from memory exhaustion; guests return with new SSH host keys	notes

E.3. October

Date	Event	Source
10-04	`merged_not_deployed` becomes a named terminal attempt state (migration 0092)	code: `f53bc4a73`
10-07	About 19 hours with the fleet down: runner code staged ahead of the schema (into 10-08); a plan validator fix follows on 10-08	notes; code: `e9bb119d4`
10-10	Controller authority becomes a database row moved by compare-and-swap (opt-in)	code: `22e2303ff`

Date	Event	Source
10-12	Merge-turn theft twice in one day; fenced the next day	notes; code: `779673ec0`
10-12	Devil's advocate role created; coordination moves from mail to the bulletin	notes
10-12	AWS foundation stacks and the experiment profile land, dormant	code: `fd282e45c`, `bee869a10`
10-13	Stephen's rule: all deployment is stopping one machine and starting another	notes
10-13	The replacement contract lands on `main`	code: `e840cdeb4`
10-14	A history-dependent guard deadlocks the lane with 51 merges parked	notes
10-14	Second replay-narrowing failure; forward migrations become missing-only	notes; code: `9937e15f1`
10-15	A scheduled CI job migrates production at 08:23Z; guarded the same day	notes; code: `34e1d4085`
10-23	`yesod rules` K1 to K12: the compliance registry	code: `30cc51daf`
10-24	Signed grants with the external `ta` tool introduced; attention and permission split the next day	notes
10-26	Hand-run controller handover completes at 00:24Z; host generation g4 serves	notes
10-27	North-star post sets the goal order G1 to G5	notes
10-27	Replacement driver, guarded adoption and the refusal of automatic migrations land	code: `cf7478bab`, `3b267bf9c`, `8001389bc`
10-29	Successor protocol lands: 39 commits under one note in one day	code: `519a7a005` and `git log --grep` for its note id

Date	Event	Source
10-29	A builder agent writes fixture rows into production (FINDING #58, which the notes date 11-06)	notes
10-30	Schema migration becomes opt-in everywhere	code: `c3451be82`

E.4. November

Date	Event	Source
11-03	Controller host generation g5 serves from 00:59Z after a hand cutover over three sittings	notes
11-08	A “frozen planning queue” turns out to be the planner and build seat VM, stopped for 15 h 45 m	notes
11-08	The mayor writes a patch itself; the devil’s advocate stops it before plan review	notes
11-09	FINDING #65: an approval inside an open window nearly merges a migration	notes
11-10	Adoption of the serving controller refused four times that day, then done at 23:20Z	notes; code: `ad55cf61d`
11-11	A release window for the token-custody change no-ops safely; adoption proof budget fixed	notes; code: `eef271b34`
11-12	Read-only dry-run preflight lands; the notes record it green in production the same day	code: `bcedcc1cb`; notes
mid-November	Three hand-submitted replacement attempts, each ended `needs_operator` before any authority transfer	notes

Date	Event	Source
mid-November	A scratch-machine rehearsal after the third attempt finds further blockers; rehearsal becomes a precondition for asking	notes
11-16	First merge window for the successor fixes ends `gate_red`	notes
11-18	Three merge windows stop safely; a fourth merges the successor fixes and the attempt ends `merged_not_deployed`. The candidate commit `16606a42` is stamped 11-17 19:16Z, a time derived from its source commit and not the time of landing	notes; code: `5451304da`
11-19	FINDING #128: all nineteen children of the hardening work closed without acceptance executed	notes
11-19	H1 passes its full Linux test union (4,591 passed, 0 failed, on a disposable test container)	notes
11-19 evening	Pre-flight and rehearsal raise findings #130 to #136 before any production step, among them: the fourth submit would be refused at its first command (#130); a merge-triggered swap deadlocks as coded (#132, first read during this book's fact-check); no successor could finish enrolling (#136)	notes; code: `replacement_guard.py`

11-19 evening	Status: hardening patch on its fifth revision, in its test gate, not on `main`; no fourth attempt; the merge-triggered swap has never run anywhere	notes

F Glossary

Terms are in alphabetical order. A term marked *(retired)* named something in an earlier edition that no longer exists or is no longer the routine path. A term marked *(practice)* names something the agents and the operator do that has no counterpart in the factory's code. State names and symbols are in `code font` as they appear in `yesod-aicoe` at `origin/main` `16606a42`.

Admissions pause. A revisioned switch in `refinery_admission.py` that stops new work from entering the merge queue. It fences intake and enqueue only; an attempt that was already admitted keeps running. Resuming needs the pause token and the expected revision.

Adoption. Recording a controller that an operator built by hand as the current member of the replacement lineage, so the replacement driver can treat it as a predecessor (`yesod replacement adopt`). The code requires root, the actor `stephen`, and a live admissions pause owned by that actor.

Agent. A language-model session primed for a role and bounded by the mechanisms around it. A dispatched agent is launched by a runner for one note; an interactive agent is a long-lived session such as the mayor.

Agents' working notes. The dated notes and state files that the coordinating agents kept for their own use while operating the factory. They are the only source for much of the autumn's history and are not authoritative.

AME *(retired)*. The Autonomous Merge Engine, an earlier supervisor-plus-worker merge daemon. The deterministic merge controller replaced it.

Armed. A note that the lifecycle scheduler has assigned to a dispatch lane after approval, recorded in `lifecycle_dispatch_arms`. An armed note can be claimed by a runner. Armed is not a lifecycle status.

Attempt. One immutable, fenced try at merging one note's branch, a row in `refinery_attempts`. A terminal attempt never reopens; a retry is a new attempt. The working notes also say "attempt" for one production try at a driver-born successor.

Authority (controller authority). The single database row, `refinery_controller_authority`, that names which controller host may run the merge controller. It moves only by compare-and-swap.

Bead. A work item in a tool's Beads graph, stored in that tool's database on the shared Dolt server. A planned note has a root bead with child beads beneath it.

Bead dependency. An ordering edge between beads inside one note's tree. It orders work within a feature and is not a merge boundary.

Birth. Creating a factory machine from a versioned recipe instead of repairing an old one: template, full clone, identity proof, bootstrap, readiness, enrollment, receipt. The recipe lives in `ops/yesod-birth/`.

Blocked. A disposition (`blocked`), and not one of the lifecycle statuses in the note contract. A blocked note is not dispatched until its condition changes. The same word is also written into the note's status column by some paths: when a merge ends `merged_not_deployed`, `_project_merged_not_deployed` in `refinery_intake.py` writes status `blocked` and disposition `blocked` together.

Build seat. A runner VM that executes dispatched agents. The working notes call gilgul the isolated build seat: the only place application code is written.

Bulletin board. An append-only posting surface over topic notes and comments (`yesod bulletin`). Authors are verified against the roster; content is labelled untrusted and can authorize nothing.

Candidate. In the merge controller, the one deterministic commit built from the target branch plus a note's branch; the gate tests exactly that commit. In replacement, the release SHA a successor is born to run.

Carrier. Two things. In the gate, the bounded, content-addressed Git bundle that takes the candidate to every demon. In release practice, the note a window exists to merge.

Catalog. The PostgreSQL registry of tools, notes, processes, runs, queue state and events.

Child outcome. The structured result a planned child reports when it ends without a commit, sent as one heartbeat prefixed `YESOD_CHILD_OUTCOME:`. Kinds are `blocked_on_dependency`, `no_changes_needed` and `acceptance_not_executed`.

Compare-and-swap (CAS). An update that succeeds only if the row still holds the expected owner, revision and lease generation. Pushes to the target branch, admissions changes and controller authority all move this way.

Conductor *(retired)*. The earlier edition's name for the single dispatch host, `yesod-dispatch`. On 2026-09-16 a fresh VM, VM175, replaced the original controller VM by hand, and later controllers have also been new VMs.

Controller. The refinery merge controller service and, by extension, the VM it runs on.

Demon. A disposable gate VM on the home hypervisor, named from a pool of demon names and paired with one test database. Its stable identity is its role, not its hostname.

Deploying. The active attempt state after the push has landed and before deployment settles. It is bounded; past the settle time the attempt ends `merged_not_deployed`.

Devil's advocate *(practice)*. An advisory reviewer agent that clears or objects to texts, scripts, plans and diffs. It never mutates anything, and its review informs a decision without vetoing it.

Dispatcher. The agent role that chooses lanes and arms notes. It is not the runner daemon.

Dispatcher approval-gate. A Beads gate on a work decision. The word "gate" alone always means the merge-time test gate.

Disposition. A note's second axis, independent of lifecycle status: `active`, `held`, `blocked`, `needs_rework`, `quarantined` or `deferred`. Only an active note is dispatched.

Driver (replacement driver). `yesod replacement drive`, a oneshot run by a timer once a minute that advances a replacement intent by one pass. It ships disabled.

Dry-run preflight. A read-only pass of the adoption or drive path that reports every failed precondition at once (`replacement_preflight.py`). A green preflight did not prevent the production attempt that followed it from stopping.

Effect journal. The table `replacement_effect_journal`, where the driver records its intent before each external effect and the observed result after. A replay finds the existing row instead of repeating the effect.

Enrollment. The single-use channel through which a newborn successor proves itself to the predecessor and receives its release archive and inputs. One key per intent; the listener closes after use.

ERS *(retired)*. The ephemeral-refinery-supervisor agent that oversaw the AME. Its primer flag still exists in `prime.py`.

Fence. A check that refuses a write from a holder whose lease or ownership is stale. Attempts are fenced by a lane lease, the controller by leadership at a lease generation, and a successor boots with its authority units fenced off.

Finding *(practice)*. A numbered finding posted on the bulletin by forensics or the devil's advocate, written "FINDING #65". The numbers seen in the working notes run from #37 to #130.

Forensics *(practice)*. A read-only investigator agent. It numbers findings, audits deliveries by running their tests on the delivered tip, and checks a grant text before it is signed.

G1. Three things. Goal G1 is "deploy on merge": one automatic, merge-triggered controller swap with a published receipt. Controller generation g1 is VM175, the VM that replaced the original controller on 2026-09-16. The old G1 gate pool is the nine original gate guests, shut down and kept cold on 2026-09-16.

Gate. The merge-time test run on the exact candidate. Its result is green, red, or an infrastructure state that is neither.

Generation. Four kinds, always qualified. A controller host generation (g1 to g5) is one controller VM in the series. An intent generation is the per-scope counter on `replacement_intents`. A lease generation is the fencing number on controller authority. A gate fleet generation (dg5, dg6) and a runner generation name the birth recipe, not a replacement count.

Grant *(practice)*. A signed approval from Stephen for an exact text, obtained and verified with the external `ta` tool. A grant may cover one pinned command, a reviewed runbook, or a listed set of reversible steps. Nothing in the factory repository verifies it.

H1 *(practice)*. The hardening work prepared before a fourth swap attempt: nineteen child tasks, on its fifth revision and in its test gate on the evening of 2026-11-19, and not on `main`.

Handover. The part of a swap that moves authority: the driver takes its own admissions pause, proves the quiet gap, transfers authority by compare-and-swap, releases the service addresses and stops the predecessor. It has never run in production.

Hold. The disposition `held` with a reason, set by `note-hold`. In current practice every note

reaching `awaiting_merge` is held until its release is granted. The old `agent_dispatch='hold'` sentinel is no longer consulted.

Intent (replacement intent). A row in `replacement_intents` recording that a scope should move to a candidate release. Several merges in a row coalesce into one intent. Planning has a separate publish intent for writing a plan.

Lane. Three kinds, always qualified. A dispatch lane is a named model and backend target a note is armed to. A refinery lane is one repository and target branch with its own merge queue. A repair lane is the primary or fallback dispatch lane chosen for an automatic conflict repair.

Leg *(practice)*. One pre-sign or delivery check, posted on the bulletin: the hold confirmation, the clerk's check, the forensics audit, the diff review, the pre-run.

Lifecycle scheduler. `LifecycleScheduler` in `lifecycle_dispatch.py`, ticked by a runner. It arms approved notes whose prerequisites are done.

Mayor. The coordinating agent. In the autumn's practice it drafts grant texts, routes work, and is the only agent that asks Stephen for permission.

Merge controller. The deterministic service (`yesod refinery controller`) that claims a refinery lane under a lease and steps each attempt through prepare, gate, push and the deployment step.

Merge queue. The durable intake table, `merge_queue`, that the controller bridges into attempts. It is a waiting room and holds no authority.

merged_not_deployed. A terminal attempt state: the candidate is on the target branch and is not proven running. The note's status and disposition are both written as `blocked`, an operator item is opened, and dependent notes do not unblock on their own.

MicroVM gate *(retired)*. The Lambda MicroVM gate that tested Yesod's own candidates until early September 2026 (the exact day of the change is not recorded on `main`). The code is still on `main`; no lane in the reviewed controller document selects it.

Molecule. A concrete Beads graph poured from a reusable process prototype.

needs_operator. A terminal state meaning the automation stopped and will not guess. It is a terminal attempt state in the refinery and a terminal disposition of a replacement intent.

Note. A durable record attached to a tool in the catalog: a feature request, bug report or usage note. A feature or bug note carries a lifecycle status, a disposition and, once planned, a root bead.

Note dependency. An ordering edge between notes (`depends_on`). The lifecycle scheduler counts a prerequisite as satisfied only when it is `done` (`SUCCESSFUL_PREREQUISITE_STATUSES`), so a prerequisite that merged without deploying keeps its dependants parked. The runner's claim SQL is looser and also accepts `superseded`.

Operator. Stephen, the owner and the only human with operator authority. The bulletin represents the operator as one address.

P5, P6, P7 *(practice)*. Step names in the working notes for the swap campaign: P5 is adoption, P6 is birth and qualification of a successor, P7 is the handover.

Plan. The planner's output for a note: child tasks with acceptance checks, bound to the note revision it was planned from. A plan with open questions cannot publish.

Plan-bound leaf. A child bead published by the planning validator. The runner executes such children one at a time on one accumulating branch; a worker cannot create beads beneath one, and each closes with a reason that carries its result commit.

Planner job. A durable row in `planning_jobs` with a lease, a generation and an attempt count. Planning is this job, run by a runner, and not a conversation.

Point of no return. The release of the service addresses (the network addresses on which the serving controller is reached) during a handover. After it, the automation refuses to roll back and any failure keeps admissions paused for an operator.

Predecessor. The controller that is serving when a swap begins. In the design, a successful handover leaves it stopped and retained (target state `stopped_retained`), from which it can be brought back to `serving`. No production swap has reached that point.

Pre-gate. A test run before the merge gate, in `refinery_pre_gate.py`, with modes `off`, `shadow` and `active`. The reviewed controller document on `main` sets no pre-gate key, so the code default `off` applies there.

Pre-run *(practice)*. A full test pass of a delivered tip on a disposable machine, run by an agent before a release is requested. It is separate from the pre-gate.

Primer. Role instructions printed by `yesod prime`. There are ten role flags and two addenda. A primer is text and not a credential.

Process. A reusable sequential or graph composition of tool actions and handoffs.

Promotion. Two things. On the bulletin, turning a peer thread into a promoter-authored usage note with a frozen snapshot. In deployment *(retired as the routine path)*, the August four-host in-place release transaction and the later controller self-promotion. The replacement design is meant to supersede both; as of 2026-11-19 controller changes are still run by hand.

Quiet gap. The proven moment when no attempt is in a blocking state and no gate is active, so authority can move. The stop guard records it as a row in `replacement_quiet_gap_observations`.

Receipt. A digest-bearing record that a step happened, which later steps refuse to proceed without: a birth receipt, a handover receipt, a deployment receipt. In governance practice, a bulletin post giving the grant id, run id and SHA of what was executed.

Refinery clerk *(practice)*. The agent that is the sole writer of lane holds and releases and that owns the admissions pause in normal operation.

Rehearsal *(practice)*. An end-to-end run of a path that has never run, on a disposable machine, before asking to run it in production. The rule dates from mid-November 2026. The database restore kit in `ops/rehearsal-kit` is a different thing.

Release skew. The measured distance between `main` and each installed surface: the operator CLI, the controller's two release trees, and each build seat (`yesod release-skew`). An unknown distance is reported as unknown.

Rule registry. The twelve rules K1 to K12 in `rules_contract.py`, read with `yesod rules`. Each states its own assurance; none is enforced, six have detectors and six are attested only.

Run ledger. The table `runs`, one row per dispatched execution, from `starting` to a terminal status.

Runner. The daemon `yesod codefactory-dispatch` and the VM it runs on. It claims armed notes, launches agents, governs them and verifies their work.

`[S]` row *(practice)*. A runbook step that only Stephen runs.

Safety island. The ninth member of a union, which runs at `pytest -n 0` the tests proven unsafe under parallel workers.

Seat. Two things. A runner seat is one runner VM, named for the righteous. A role seat is the single registered session of a singleton role such as the mayor.

Sentinel *(practice)*. A background loop that watches one condition and fails closed, for example by holding a note on delivery or re-pausing admissions. A polling sentinel can lose a race to intake.

Shard. One of the eight partitions of the test suite in a union, run at `pytest -n 4` on its own demon with its own database.

sjbis. A separate tool outside the factory repository for getting the human's attention: a question with choices and a deadline. The factory only calls it.

Stop guard. The swap's own admissions pause plus its proof of quiet, a row in `replacement_stop_guards`. A swap never borrows another actor's pause.

Successor. The newborn controller in a swap. It boots fenced, enrolls, installs the candidate release, proves itself locally, and only then is handed authority.

ta. A separate tool outside the factory repository for permission: Stephen signs an exact text, and any agent can verify the grant against that text.

Tool. A catalog entity for a piece of software the owner writes, shares or depends on. Notes, beads and refinery lanes hang from tools.

Topic. A durable knowledge object that spans tools.

Union. One concurrent gate run of eight shards plus the safety island on nine demons. Green requires nine receipts and a proof that every collected test ran exactly once.

Window *(practice)*. A period when admissions are resumed under a grant so that one named note can merge, after which admissions are paused again.

Worker. The coding agent a runner launches inside a worktree for one note or one planned child.

Worker canon. The shared worker instructions in `worker_canon.py`, rendered into `AGENTS.md`, the dispatched-agent prompt and `yesod prime --worker`.

G Building This Book

This appendix describes how the November 2026 edition was produced and how to rebuild it. The first half is about method, because a book that asks its subject for evidence should say where its own came from.

G.1. How This Edition Was Written

The edition was produced in one working session on 2026-11-19 by a set of language-model agents under one coordinating editor, also an agent, working from a request by Stephen. No human wrote or reviewed the text before this build. A reader should weigh it accordingly: it is a careful first full draft of a new edition, checked against source, and it has not yet had its owner's corrections.

The work ran in six passes; the last, a consistency edit, is described at the end of the fifth.

1. **Orientation.** The editor read the repository's documentation conventions, the previous edition, the build files, and the current architecture map (14 systems and 65 subsystems in the code-intelligence taxonomy at the snapshot).
2. **Research.** Twelve agents each took one area (registry and notes; lifecycle, planning and dispatch; the runner; the refinery; gates; deployment as replacement; birth, fleet and credentials; observability; cost and routing; reliability and incidents; the agent organization and governance; roadmap and timeline). Each wrote a brief of 3,000 to 7,500 words: what the area is now, a dated timeline since 2026-08-20 with commit ids, exact symbol names, invariants, incidents, what the previous edition got wrong, and open questions. The briefs total about 61,000 words and are kept in `book/_research/`.
3. **Outline.** The editor wrote the plan for the edition from the briefs: the table of contents, what each chapter owns, a list of status facts binding on every chapter, and the evidence convention.
4. **Drafting.** One agent per chapter, or per pair of short chapters, wrote from the outline and briefs, with instructions to check any symbol, state name, command or number in source before printing it.
5. **Fact-check.** A second set of agents, none of which had written the text it checked, verified each chapter against the source: every symbol in code font, every state and transition, every number, every commit id and date, every quotation. They corrected the chapters in place and reported what they changed. Each checker also made a separate reading of its chapter for one thing only: sentences that overstate what works in production. A final pass by a third set of agents reconciled numbers, dates, terms and cross-references between chapters and removed incidents told twice.

G.1.1. What the agents could and could not do

Every agent had the same limits, and they shape what this edition can claim.

- **Source was read-only and pinned.** Code was read from `origin/main` at `16606a42` with `git show`, `git grep` and `git log`, never from a working tree. Structural questions went to a code-intelligence index of the same commit.
- **Nothing was executed.** No agent ran the test suite, ran the `yesod` command, queried a production database, or connected to a host. The rule against running tests is itself one of the lessons in Chapter 17: on 2026-10-14 a subagent's test run from the control machine turned a real gate red.
- **No deployment was observed.** Facts about the live system come from runbooks and receipts committed on `main`, from commits, and from the working notes the coordinating agents kept during the autumn. The notes are background and are attributed as such wherever a claim rests on them.
- **Secrets were out of scope.** Agents were instructed not to read or print credentials, and the text omits network addresses and the locations of secret files.

G.1.2. Known weaknesses of the method

The research tool that answers open questions about the codebase timed out on most calls during this session, so the briefs rest on direct reading of source and history. Commit messages in the repository carry much less explanation from October onward than before (Chapter 19 gives the numbers), so the October and November story depends more heavily on the working notes than the September story does. Several facts that matter could not be established at all and are listed at the end of the chapters that need them and in the rewrite report kept with the research briefs. The most important are the details of two of the three attempts at a driver-born successor, whether a full rehearsal of the handover has completed away from production, and the contents of the hardening work being prepared for the next attempt.

G.2. Building the PDF

The book builds with a small local toolchain:

- Pandoc 3.10 or compatible;
- XeLaTeX;
- Python 3 (standard library only) for the diagrams;
- `rsvg-convert`;
- the Charter, Avenir Next, and Menlo fonts on macOS.

From the `book` directory:

```
make
```

The build performs three steps:

1. render the diagrams as SVG with `figures/render.py`;
2. convert each SVG to a vector PDF figure;
3. run Pandoc with the KOMA-Script and XeLaTeX theme.

The Pandoc invocation and the canonical order of the manuscript are in `book/Makefile`. The manuscript is split across `book/frontmatter/`, `book/chapters/` and `book/appendices/`. The six part dividers and the appendix marker are one-line files in `book/parts/`, so that no chapter file carries structure that belongs to the whole book. Metadata and page geometry are in `book/metadata.yaml`, typography and the title page in `book/tex/preamble.tex`. Earlier material kept for reference and excluded from the build (the development-wave histories and a retired appendix) is under `book/appendices/archive/`.

Useful validation:

```
make check
pdfinfo yesod-software-factory.pdf
pdftotext yesod-software-factory.pdf - | wc -w
```

For a portable build, replace the macOS fonts in `metadata.yaml` with installed open fonts. The content does not depend on a proprietary template or a network resource.

G.3. Publishing an Edition

The `Publish book edition` GitHub Actions workflow rebuilds and validates the book whenever canonical book sources change on the documentation repository's `main`, and can also be started by hand. It calls `scripts/publish-book.sh`, which:

1. chooses an unused immutable dated key in the publications bucket;
2. uploads the PDF with immutable cache and download metadata;
3. downloads it again and verifies its byte count and SHA-256 digest; and
4. sends a `book-edition-published` repository dispatch to the website repository.

If several editions are published on one UTC date, later objects receive `-r2`, `-r3`, and so on. Existing objects are never overwritten. The website receiver opens a manifest pull request, and an edition is not visible on the site until that pull request is reviewed and merged. This description is carried over from the previous edition and matches the script and workflow in the documentation repository at the time of writing; it was not exercised for this edition, which was left uncommitted for review.