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 · Economics · Future

Source-grounded edition · August 17, 2026

Revision fc8fbca

Stephen

Contents

About This Edition **1**
How to Read This Book 2

Preface: A Factory, Not a Chat Room **3**

Prologue: The Accidental Factory **4**

I. One Change Through the Factory 6

1. Follow the Change **7**
1.1. One Door, Precisely Stated 10
1.2. The Proof Chain 10

2. Judgment and Mechanism **12**
2.1. The Control Plane 12
2.1.1. Bootstrapping a role with `yesod prime` 13
2.1.2. The planning–dispatch boundary 15
2.2. The Execution Plane 15
2.3. The Integration Plane 15
2.4. The Data Plane 16
2.5. Why Determinism Matters 16

3. The Two Currencies **18**
3.1. Notes: Intent and Dispatch History 18
3.2. Beads: Work and Dependency Structure 19
3.3. Two Dependency Layers 19
3.4. Lanes, Hosts, and Affinity 20
3.5. Three State Machines, Not One 21

4. The Registry Beneath the Factory **22**
4.1. Tools as Durable Context 22
4.2. Querying One Tool in Its Ecosystem 23
4.3. Processes as Molecules 23
4.4. Composition Creates a Feedback Network 24
4.5. Topics: Knowledge Without a Tool Owner 25
4.6. More Than a Factory Front End 25

II. From Intent to Merge **27**

5. Planning and Dispatch **28**
5.1. Capture Before Decomposition . . . 28
5.2. Complexity Is Routing Data . . . 29
5.3. Choose Merge Boundaries Before Task Order . . . 29
5.4. Pre-flight Questions . . . 30
5.5. Arming . . . 30

6. The Runner **32**
6.1. The Ten-Second Loop . . . 32
6.2. The Claim Boundary . . . 33
6.3. Worktree Isolation . . . 33
6.4. Timers That Define Failure . . . 34
6.5. The Run Ledger . . . 34

7. Governed Agent Execution **35**
7.1. The Worker Canon . . . 35
7.2. Governance Around the Process . . . 36
7.3. What Counts as Success . . . 36
7.4. Failure Has Two Axes . . . 37
7.5. Bounded Retries . . . 37

8. The Merge Controller **38**
8.1. The Vocabulary of a Merge . . . 39
8.2. The Good Case . . . 40
8.3. When the Gate Says No . . . 40
8.4. When the Owner Stalls . . . 41
8.5. When `main` Moved Underneath . . . 41
8.6. When Another Change Wants the Same Branch . . . 42
8.7. When the Merge Itself Conflicts . . . 42
8.8. When the Door Is Closed . . . 43
8.9. When the Process Simply Dies . . . 43
8.10. The Tick, Assembled . . . 43

9. The Deterministic Gate **45**
9.1. The Vocabulary of a Gate . . . 45
9.2. The Good Case . . . 46
9.3. When the Tests Fail . . . 47
9.4. When the Gate Cannot Run . . . 47
9.5. Why a Disposable Machine at All . . . 48
9.6. When Disposable Infrastructure Leaks . . . 48
9.7. When the Safety Net Reads the Wrong Contract . . . 49
9.8. When the Image Is Stale . . . 49

10. Merge Is Not Deploy **50**
10.1. The Vocabulary of a Promotion . . . 50

10.2. The Good Case . . . 51
10.3. When a Host Is Still Busy . . . 52
10.4. When a Proof Fails After Activation . . . 52
10.5. When a Proof Fails Before Activation . . . 53
10.6. When the Worker Itself Crashes . . . 53
10.7. The Part That Is Still Each Tool's Contract . . . 53
10.8. Proving Delivery . . . 54

11. Seeing the Whole System 55
11.1. The Observation Decision . . . 55
11.2. The Existing Evidence Map . . . 56
11.3. The Factory Event Envelope . . . 56
11.4. Coverage Without Invented History . . . 57
11.5. Human and Agent Read Models . . . 58
11.6. OpenTelemetry Projection . . . 58
11.7. Private Content and Network Authority . . . 59
11.8. Current Windows Into the Factory . . . 59
11.9. Temporary SigNoz Deployment . . . 59
11.10. Delivery Program and Evidence . . . 63
11.11. What the First Live Days Taught . . . 65
11.12. A Read-Only Diagnostic Rhythm . . . 66

12. Failure Is a State Machine 67
12.1. The Vocabulary of Failure . . . 67
12.2. Execution Failures — Where the Work Is Made . . . 68
12.3. Integration Failures — Where the Work Lands . . . 68
12.4. Deployment Failures — Where the Work Runs . . . 69
12.5. A Note on Host Placement . . . 69
12.6. The Shape of the Whole Machine . . . 70

13. Cost-Aware Dispatch 71
13.1. Capturing Usage Honestly . . . 71
13.2. The Wrong Metrics . . . 71
13.3. Cost Awareness Today . . . 72
13.4. Expected Delivered Cost . . . 73
13.5. Bounded Spend Is Part of Routing . . . 73
13.6. Project-Specific Policy . . . 73
13.7. Exploration Without Gambling the Factory . . . 74
13.8. The Next Cost-Aware Step . . . 74

III. Building Beyond the Current Factory 75

14. Patterns Worth Reusing 76
14.1. Durable Intent Before Automation . . . 76
14.2. Separate Work Graph from Execution History . . . 76
14.3. Make Dispatch Orthogonal . . . 76

14.4. Claim Atomically, Repair Cross-Store Effects . . . 76
14.5. Give Every Run a Disposable Workspace . . . 77
14.6. Put Governance Outside the Model . . . 77
14.7. Define Proof of Work . . . 77
14.8. Use One Automated Integration Door . . . 77
14.9. Distinguish Red from Unavailable . . . 78
14.10. Reconcile Durable Scratch . . . 78
14.11. Record Reasons, Not Only Decisions . . . 78
14.12. Price Delivered Outcomes . . . 78
14.13. Make Authority a Data-Flow Property . . . 78
14.14. Date the Fleet . . . 79
14.15. An Adoption Ladder . . . 79

15. The Reliability Contracts 80
15.1. The Simple Example That Started It . . . 80
15.2. The Five Rules, Named . . . 81
15.3. Recovery Is a Contract Too . . . 81
15.4. The Campaign: Eleven Attempts, Zero Corruptions . . . 82
15.5. Where the Evidence Lives . . . 84
15.5.1. One Row, Read Closely . . . 84
15.6. What a Reliability Contract Is . . . 85

16. The Future of Yesod 86
16.1. A Typed Tool Ecology . . . 86
16.2. Processes as a Workflow Compiler . . . 86
16.3. Reciprocal Tool Improvement . . . 87
16.4. An Adaptive Dispatch Economist . . . 87
16.5. A Factory Digital Twin . . . 88
16.6. A Reliability Immune System . . . 88
16.7. Signed Provenance from Intent to Runtime . . . 89
16.8. Multi-Repository Change Sets . . . 89
16.9. First-Class Upstream Contribution . . . 90
16.10. Living Documentation . . . 90
16.11. What Yesod Should Not Become . . . 90
16.12. A Plausible Sequence . . . 91

Epilogue: Scar Tissue 92

A. Command Field Guide 93
A.1. Catalog and Knowledge . . . 93
A.2. Processes and Molecules . . . 93
A.3. Work Graphs . . . 94
A.4. Factory Status . . . 94
A.5. Dispatch and Runs . . . 94
A.6. Refinery . . . 95
A.7. Propulsion . . . 95

B. States and Timers **96**
B.1. Note State . . . 96
B.2. Dispatch State . . . 96
B.3. Run State . . . 97
B.4. Merge Queue State . . . 97
B.5. Timing Reference . . . 97

C. Roles and Authority **99**
C.1. Agent Roles . . . 99
C.2. Deterministic Services . . . 99
C.3. Authority Rules . . . 100

D. Evidence and Source Map **101**
D.1. Current Deployment Snapshot . . . 102
D.2. Documentation Corrections Incorporated Here . . . 102

E. Yesod Development Waves **104**
E.1. Wave 1: The Self-Hosting Exit . . . 104
E.1.1. Wave Summary . . . 104
E.1.2. The Cutoff Was a Feature . . . 105
E.1.3. What Shipped . . . 105
E.1.4. A MicroVM Became a Managed Gate . . . 107
E.1.5. Evidence at the Exit . . . 107
E.1.6. Ready Does Not Mean Configuration-Free . . . 108
E.1.7. The New Metric . . . 109
E.2. Wave 2: The Async Gate and the Operator Surface . . . 109
E.2.1. Wave Summary . . . 109
E.2.2. The Gate Became a Job Protocol . . . 110
E.2.3. The Cutover Failed at the Image Boundary . . . 110
E.2.4. The Operator Stopped Needing `psql` . . . 111
E.2.5. Observability Learned to Name States . . . 112
E.2.6. What Shipped, and What Stayed Out . . . 112
E.2.7. The New Exit Criterion . . . 113
E.3. Wave 3: The Crash-Only Factory . . . 114
E.3.1. Wave Summary . . . 114
E.3.2. Readiness Became a Control-Plane Fact . . . 115
E.3.3. The Refinery Became Crash-Only . . . 115
E.3.4. The Fleet Survived Its First Real Storm . . . 116
E.3.5. A Generic Gate Acquired a Repository Contract . . . 117
E.3.6. The Factory Learned to Create Projects and Agents . . . 117
E.3.7. The Dashboard Became a Deployable Surface . . . 118
E.3.8. What Shipped, and What Stayed Out . . . 118
E.3.9. The New Exit Criterion . . . 119
E.4. Wave 4: Integration Branches and Bounded Intelligence . . . 119
E.4.1. Wave Summary . . . 119
E.4.2. A Note Can Choose Its Integration Branch . . . 120
E.4.3. `main` Stayed Releasable . . . 121

E.4.4. Distillation Became a Bounded Maintenance Job . . . 122
E.4.5. A Stall Now Pages a Human . . . 122
E.4.6. The Proxied Dashboard Became Complete . . . 122
E.4.7. What Shipped, and What Stayed Out . . . 123
E.4.8. The New Exit Criterion . . . 123
E.5. Wave 5: Dogfooding the Factory on TTRAC . . . 124
E.5.1. Wave Summary . . . 124
E.5.2. The Demo Had a Real Dependency Shape . . . 125
E.5.3. Dispatch Became Observable Product Behavior . . . 125
E.5.4. The Demo Found a Gate That Was Too Narrow . . . 126
E.5.5. Operations Were Part of the Proof . . . 126
E.5.6. What Shipped, and What Stayed Out . . . 126
E.5.7. The New Exit Criterion . . . 127
E.6. Wave 6: Identity, Idleness, and the Quiet Factory . . . 127
E.6.1. Wave Summary . . . 127
E.6.2. Idle Is a Derived State, Not an Empty Screen . . . 129
E.6.3. Coordination Has a Lifetime . . . 129
E.6.4. The ALS Incident Exposed an Identity Boundary . . . 130
E.6.5. The Server Became Canonical . . . 131
E.6.6. One Populated Database Still Needs a Deliberate Migration . . . 131
E.6.7. What Has Landed, and What Remains . . . 132
E.6.8. The New Exit Criterion . . . 132
E.7. Wave 7: The Legible Factory . . . 133
E.7.1. Wave Summary . . . 133
E.7.2. The Factory Had State but No Story . . . 134
E.7.3. Parked Work Is Not Rejected Work . . . 134
E.7.4. Grounding Became a Durable Asset . . . 135
E.7.5. From Event Rail to Activity Stream . . . 136
E.7.6. Attention Became a Control Plane . . . 137
E.7.7. What Has Landed, and What Remains . . . 137
E.7.8. The New Exit Criterion . . . 138
E.8. Wave 8: The Conductor and the Deterministic Refinery . . . 139
E.8.1. Wave Summary . . . 139
E.8.2. The Run Began With a Real Pause . . . 141
E.8.3. An Attempt Became the Unit of Truth . . . 141
E.8.4. External Effects Became Reconciled Facts . . . 142
E.8.5. Review Found Two Missing Invariants . . . 142
E.8.6. Activation Required More Than Green Tests . . . 143
E.8.7. The Gate Is Remote; the Controller Is Not the Test Host . . . 143
E.8.8. The Workstation Stopped Being the Server Room . . . 144
E.8.9. The Conductor Has an Explicit Filesystem Contract . . . 145
E.8.10. Standing Services Moved as a Set . . . 146
E.8.11. Human Communication Became Standing Infrastructure Too . . . 146
E.8.12. Cutover State Was Reconciled, Not Waved Away . . . 147
E.8.13. Two Final Alerts Improved the Design . . . 147
E.8.14. A Third Lane Proved the Whole Terminal Boundary . . . 148

E.8.15. Onboarding Became an Idempotent Command . . . 149
E.8.16. The Refineries Page Became Self-Explaining . . . 150
E.8.17. What Is Live . . . 151
E.8.18. The Exit Criterion Was Met . . . 151
E.9. Wave 9: Universal Intake and the Project Fleet . . . 152
E.9.1. Wave Summary . . . 152
E.9.2. The Starting Point Was Three Proven Lanes, Not a Fleet . . . 153
E.9.3. Fleet Membership Became Reviewed Policy . . . 153
E.9.4. Onboarding Became One Idempotent Operation . . . 154
E.9.5. Coding Gates and Merge Gates Are Different Contracts . . . 154
E.9.6. Twenty-Two Sentinels Tested the New Lanes . . . 155
E.9.7. A macOS Lock Was Not a Safe Linux Gate . . . 156
E.9.8. A Declared Linter Is Not an Installed Linter . . . 156
E.9.9. Terminal Projection Needed Its Own Controller Workspace . . . 156
E.9.10. Placement Did Not Drift During Expansion . . . 157
E.9.11. Interim Disk Headroom Without a Placement Change . . . 158
E.9.12. The Operational Commands Are Boring on Purpose . . . 158
E.9.13. What Is Live . . . 159
E.9.14. The Exit Criterion . . . 159
E.10. Wave 10: Lifecycle v2 and the Self-Governing Factory . . . 160
E.10.1. Wave Summary . . . 160
E.10.2. The Problem Was That `open` Meant Too Many Things . . . 161
E.10.3. The CLI Became Explicit Without Becoming Fussy . . . 162
E.10.4. Plans Became Revision-Bound Durable Objects . . . 162
E.10.5. Deterministic Promotion and Dispatch Replaced Remembering . . . 162
E.10.6. Implementation Was a Program, Not an Enum Patch . . . 163
E.10.7. Production Found the Bugs Unit Tests Could Not . . . 163
E.10.8. The Cutover Was a Sequence of Capabilities . . . 165
E.10.9. The Canary Went Through the Real Factory . . . 165
E.10.10. Legacy Migration Was Conservative . . . 166
E.10.11. What Lives Where Now . . . 166
E.10.12. The Operator Surface Shows Real Capacity . . . 167
E.10.13. Final State . . . 168
E.10.14. The Exit Criterion . . . 168

F. Gas Town, Gas City, and Yesod 170
F.1. Research Snapshot . . . 170
F.2. The Central Distinction . . . 171
F.3. Where the Roles Live . . . 172
F.4. Planning and Decomposition . . . 172
F.5. Claim, Execution, and Isolation . . . 173
F.6. Integration Is the Clearest Divergence . . . 173
F.7. Recovery and Health . . . 174
F.8. Parallelism . . . 174
F.9. Durable State and Evidence . . . 175
F.10. Cost and Model Selection . . . 175
F.11. Knowledge Beyond Work Tracking . . . 176

F.12. Authority and Trust 176
F.13. What Yesod Should Borrow 176
F.14. What Yesod Should Keep 177
F.15. A Practical Integration Path 177
F.16. Sources 178

G. Glossary **179**

H. Building This Book **181**
H.1. Publishing an edition 181

About This Edition

This book describes **Yesod**, an autonomous software factory that grew out of a personal tool registry. It is not a product brochure and it is not a dump of command help. It is a source-grounded account of how durable intent becomes planned work, how bounded coding agents turn that work into branches, how a deterministic merge controller decides what may reach `main`, and how the result becomes running software.

The source snapshot for this edition is:

Item	Snapshot
Repository	`yesod-aicoe`
Baseline commit	`4d658cc6` (`origin/main`, 2026-08-16 UTC)
Re-verification date	2026-08-16 PDT
Prior-edition baseline	`cf091d2e…` (2026-07 snapshot)
Deployment observation	four-host fleet promoted to release `479f1113`, 2026-08-16
Documentation project	`yesod-aicoe-docs`

This edition re-grounds the merge, gate, and deployment chapters on the **reliability redesign**. The standing LLM-based ephemeral merge supervisor was retired in favor of a deterministic **merge controller** built on lanes, immutable attempts, and leases; the remote gate became a provider-neutral, exact-candidate contract; and Yesod's own deployment became a durable four-host **promotion transaction**. Chapters 8–10, 12, and 17 were rewritten against the current source; the earlier registry, planning, dispatch, and execution chapters were re-verified and corrected wherever the redesign changed a mechanism or a command. Where a claim reflects a runtime observation rather than committed code — a release SHA, a fleet count, a campaign event — it is dated and marked as such.

Facts in this book follow a strict precedence:

1. **Code** — behavior verified in the pinned source tree.
2. **Deployment** — read-only observation of the live fleet, dated because it can drift without a commit.
3. **Historical** — repository history and earlier design documents, used to explain how the system evolved.
4. **Design** — intent or planned work that is not yet an operational guarantee.

That distinction matters. A service can be designed for one host and run on another. A status can be declared in an enum but never written by the current automated path. A safety

control can exist in source while the deployed process still runs older code. In a software factory, documentation that erases those differences creates operational risk.

The main edition's source claims remain pinned to the snapshots above. Chapter 11 now also carries a clearly marked design addendum for the planned Observation Framework and a separately dated observation of its temporary diagnostic backend. Credentials, private addresses, and other secrets are intentionally omitted.

How to Read This Book

Parts I and II trace one change through the system. If you are new to Yesod, read them in order. Part III is an operator's guide to evidence, recovery, cost, and throughput. Part IV extracts patterns that can be reused in other agentic systems. The appendices provide commands, timings, terminology, and a source map.

Five recurring sidebars keep the argument honest:

- **Invariant** — a property the system is built to preserve.
- **Terminology Trap** — a word whose casual use hides two different mechanisms.
- **Operator Check** — a safe, read-only way to establish current state.
- **Factory Floor** — an incident that changed the design.
- **Design Trade-off** — a deliberate exchange of speed, cost, safety, or complexity.

The commands are examples, not invitations to operate production blindly. Read-only status commands are shown freely. Mutating commands appear only when needed to explain a contract and should be governed by the system's authority rules.

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?

Yesod is the collection of answers to those questions.

Its central move is a separation of concerns:

- **Agents make judgments.** They clarify intent, inspect source, decompose work, choose a lane, diagnose failures, and propose 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 is more important than the number of models or hosts. Without it, an agent's fluent assertion becomes indistinguishable from a completed action. With it, every important claim has a mechanical proof: a locked row, a commit past a base SHA, a verified remote branch, a gate return code, a pushed `main`, a deploy outcome, or a durable telemetry record.

> **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 not infallible. This edition documents real seams: declared statuses that are not used by the current path, legacy exact child-note compatibility beneath a cleaner feature-note planning contract, and an SPS watchdog deployed on a different host from the process it intends to signal. The point of an advanced system description is not to hide those facts. It is to make them legible.

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. The system gained a graph of work as well as a 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. A live graph made it possible to watch the dependency structure move.

On June 1, commit `14004657` added `codefactory-dispatch`. A note could be assigned to a model lane; 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.

The next six weeks were an exercise in scar tissue.

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 no-progress results. 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, compatibility-aware batching, a global merge lock, branch verification, per-note red isolation, retry caps, merge summaries, and post-merge hooks. The gate moved from the operator workstation to disposable MicroVMs, shortening the critical path but adding 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, not value. 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**, not cost per million tokens in isolation.

By July 11 the repository contained more than two thousand commits, roughly 97,000 lines of Python under `yesod/src/yesod`, 236 Python test modules, a multi-repository refinery registry, a live fleet, and a remote gate with explicit storm controls. None of that was present in the original roadmap because there was no such roadmap.

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.

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 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. The agent carries broad knowledge, changes roles fluidly, and keeps the whole feature in one conversation. For a solo project, an experiment, or a low-risk application, that may be exactly the right operating model.

But look closely at the job Maya has given the agent. In one context window it must be the product planner, systems designer, implementer, test author, reviewer, release manager, and incident historian. Every new responsibility pulls the same attention in another direction. Product assumptions sit beside stack traces. An early design choice remains in the context while the agent later evaluates 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, an interface, tests, documentation, or an infrastructure adjustment. One agent in one session normally walks that path serially. Maya can open more terminals, but then she becomes the scheduler, assigns scope by hand, keeps dependencies in her head, and reconciles the results herself.
2. **Focused, role-based agents.** Planning, implementation, testing, triage, integration, and operations reward different kinds of attention. A planner should clarify uncertainty before code exists. A worker should stay inside a bounded task. A tester should search for counterexamples rather than defend the implementation. An integrator should protect `main`, not preserve the worker's sense of progress. Giving each function a clean context and a narrow contract lets it become better at its own job.

The missing ingredient is **productive tension**. In a single-agent session, one fluent line of reasoning can carry a feature from idea to merge. There may be self-critique, but there is no durable boundary at which another role must accept the evidence. The same mind proposes the plan, makes the trade-offs, and judges the result.

A factory creates tension without creating chaos. The planner pushes for explicit scope. The worker pushes for an implementable change. The test gate tries to falsify the worker's claim. The integrator protects the shared branch. The dispatcher weighs urgency, capability, and cost. Operations asks whether merged code actually became running code. These functions do not need to be adversaries, but their incentives should not collapse into one unexamined conversation.

> **Design Trade-off — From acceleration to organization.** Vibe coding makes a great generalist faster. A software factory adds parallel work, specialized attention, independent proof boundaries, and durable coordination around that generalist capability.

Yesod begins at that boundary. It does not merely ask one model to impersonate an entire team in a longer prompt. It gives distinct agent sessions bounded roles, represents work and dependencies outside their context windows, and places deterministic services between judgment and authority. Agents may plan, implement, investigate, or supervise; mechanisms decide what is claimable, what counts as progress, and what may merge.

The easiest way to see the difference is to follow one change through the factory.

Now imagine the equivalent request inside Yesod itself: add a read-only command that explains why an armed note is not running. The exact feature is less important than the artifacts it creates.

The request first becomes a **note**. A work note represents one independently mergeable feature and carries its durable narrative and dispatch record: what should change, why it matters, what evidence would count, which root Bead it belongs to, and which lane—if any—may execute it. Work notes live in PostgreSQL and carry a lifecycle status.

The request is then grounded into **Beads**. The note links to one root Bead; its children form the internal checklist for source changes, tests, and documentation that should land with that feature. Bead dependency edges express ordering inside the checklist. The Beads graph lives in a per-tool workspace backed by Dolt.

The distinction is deliberate:

- the note answers **which mergeable feature are we dispatching, what must land before it, and what happened to it?**
- the Bead tree answers **how is this feature decomposed, and what checklist step is ready next?**

When the plan is ready for dispatch, a note is **armed** by assigning a lane to its `agent_dispatch` field. Arming is not a lifecycle state. The note can be `open` and unarmed, `open` and armed, or explicitly held. A lane names a configured execution target—a model and the command used to invoke it—not a host.

A runner daemon polls for the intersection of two sets:

1. work notes that are open and armed and pass note-level dependency, routing, and retry checks;
2. beads that are dependency-ready in the correct per-tool workspace.

The runner claims the armed feature note linked to the root and uses the Bead tree to execute ready checklist children in dependency order on one candidate branch. A child that truly

needs an independent branch or merge decision should already have been promoted during planning to a separate note and root Bead, with a note-level dependency connecting the features.

The claim is a short database transaction with row locking. Only after the transaction wins does the runner assign the bead. If that later assignment fails, the runner releases the note claim. That is the first proof boundary: two runners may see the same candidate, but they cannot both own the same note.

The runner creates a Git worktree, launches the selected coding agent, and governs the process. The worker reads the actual source, commits incrementally, runs targeted checks, and records progress events. It never pushes `main`.

When the process exits, the runner does not take the transcript's word for success. For ordinary code work it looks for commits past the base SHA. It executes any declared acceptance and terminal user interface (TUI) smoke checks. An explicit failing check blocks progress; an absent or skipped check is not the same as a failure. For `task_kind=data`, the contract is stricter in another direction: the acceptance probe replaces the commit requirement and must pass.

The runner then force-pushes a named remote branch and verifies that the branch exists. The feature note advances to `awaiting_merge`. Queue enrollment is a separate durable operation, not the same transaction as the status change; periodic reconciliation repairs the gap if a process dies between them.

The **merge controller**—the deterministic core of the refinery—takes over. Each tick it claims the note's lane under a lease, builds an exact merge candidate against current `origin/main`, and runs the repository profile's gate against that exact candidate. There is no batch to assemble and no offender to isolate: one attempt owns a lane at a time, and a retry is a new, freshly-fenced attempt rather than a reopened one.

On green, it pushes the exact gated candidate to `origin/main`, verifies the push, and verifies that each note's commits actually landed. It writes durable merge evidence and advances notes to `done`. But `done` means *source-complete*, not deployed: for Yesod itself, making the fleet run the merged code is a separate four-host promotion transaction (Chapter 10), and for other tools it is whatever their profile's deploy action declares.

On a red gate result, the worker does not push. For a multi-note batch it isolates notes to find the offender. Innocent notes can land while the offending note's queue row records a gate attempt. Infrastructure failure is different again: if the remote gate cannot run, the gate attempt is not consumed and the work remains queued.

The whole path can be compressed into one sentence:

> **Intent becomes a note and a bead graph; arming makes eligible work visible to runners; a runner produces a verified candidate branch; the merge controller gates and lands it; and delivery to the running fleet is proven separately, because merge is not deploy.**

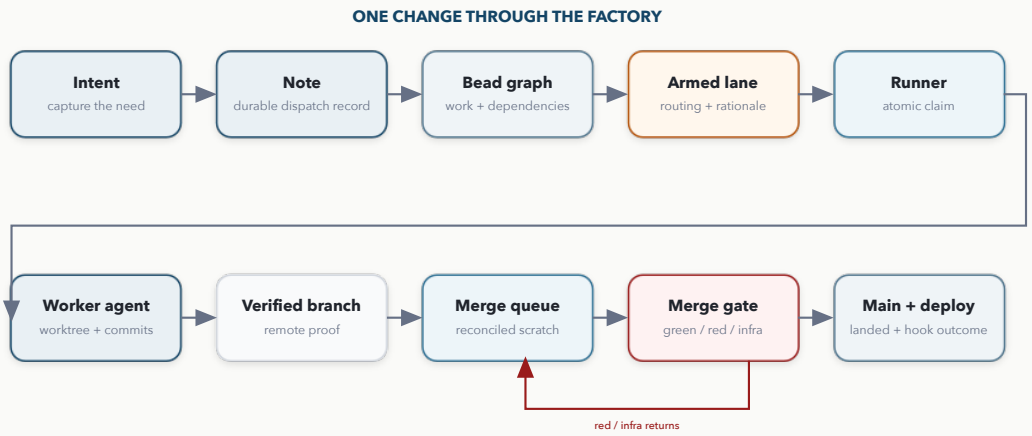


Figure 1.1.: One change moving through Yesod.

1.1. One Door, Precisely Stated

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

> **Invariant — Intended automated path.** For owned repositories managed by the factory, the merge controller is the automated component authorized to update `origin/main`. It pushes only a candidate tree that has passed the profile gate; after the push, it verifies the remote SHA and each note's commit ancestry before recording the notes as landed.

This is an architectural invariant, not a claim that Git hosting makes all other credentials physically impossible. A human with repository authority can still act outside the factory. Documentation should not confuse a controlled system path with an absolute property of the remote server.

1.2. The Proof Chain

Each stage produces evidence for the next:

Stage	Evidence
Planning	note–bead linkage and dependency graph
Dispatch	audited lane assignment and reason
Claim	locked note row, runner ownership, bead assignment
Execution	run ledger row, log, progress events, commits
Verification	acceptance/TUI result and remote branch proof

Stage	Evidence
Integration	merge job, gate run, pushed SHA, landed-commit check
Post-merge action	structured hook outcome and log

The last row proves what the hook reported, not necessarily that users can exercise the code; Chapter 10 returns to that distinction. More broadly, this chain is the difference between an agent demo and a factory. A demo can end when the model prints a confident summary. In a factory, every handoff must carry durable evidence that the receiving stage can verify.

The next chapter separates the kinds of judgment, mechanism, and state that produce this evidence.

2 Judgment and Mechanism

Yesod divides the system into four conceptual planes.

The control plane assigns judgment, the execution plane governs coding runs, the integration plane controls entry to `main`, and the data plane preserves durable state and evidence.

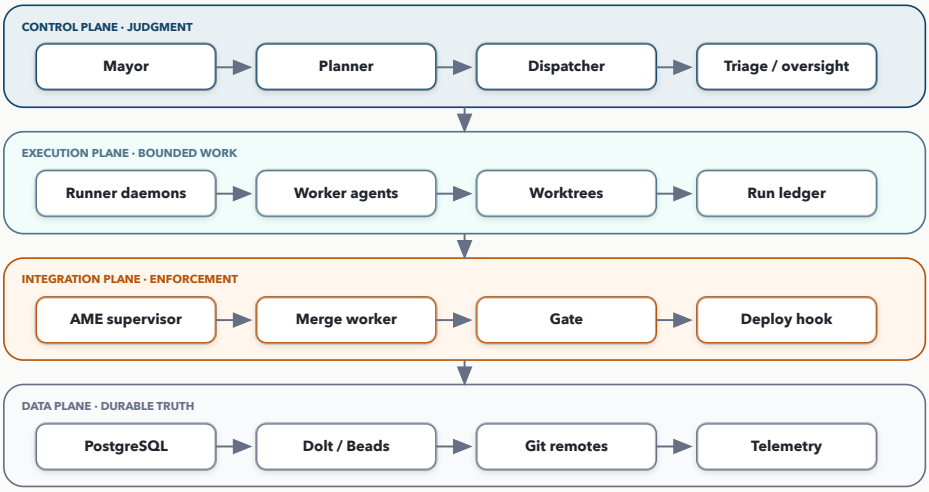


Figure 2.1.: The four planes of the factory.

2.1. The Control Plane

The control plane is where Yesod puts decisions that cannot be reduced to a safe mechanical rule. Is a request specific enough to plan? Which work can proceed independently? Does a failure point to the feature, the agent, or the infrastructure? Is an expensive model justified? These are judgments under uncertainty, so Yesod assigns them to agents. It does not, however, ask every agent to make every kind of judgment.

A **role** is a bounded operating contract for an agent session. It defines the outcome that session should optimize, the evidence it should consult, the actions it may take, the actions it must not take, and the next role to which it should hand work. The role is more than a job title or a theatrical persona. It is a way to keep one context focused and to make its responsibilities legible to the rest of the factory.

This separation creates the productive tension introduced in Chapter 1. A planner is rewarded for grounding and decomposing intent, not for rushing into implementation. A dispatcher is rewarded for sending ready work to an appropriate lane, not for rescuing an incomplete plan. Triage is rewarded for finding the actual failure boundary, even when that means rejecting the first explanation. Infra-monitor is rewarded for preserving fleet safety rather than maximizing short-term throughput. Each role sees the same factory from a deliberately different angle.

Roles are also independent of model identity. The same underlying model can occupy different roles in different sessions; what changes is its operating context and mandate. Conversely, calling a session a planner does not grant it authority or make its output true. The planner still has to produce durable artifacts that later roles and deterministic services can inspect.

Across the factory, the current registry exposes ten role primers:

Role	Primary focus
mayor	serve as the human-facing lead and route work
planner	read source and turn intent into executable structure
dispatcher	choose where and when planned work should run
triage	investigate failures and locate the actual failure boundary
infra-monitor	reason about hosts, staging, and fleet safety
mad-scientist	run controlled, measured experiments
overlord	manage the fleet of long-lived roles
ephemeral-refinery-supervisor (ERS)	provide judgmental oversight of the merge system when present
worker	define the contract for a runner-launched coding agent
refinery	define the contract for a human-driven refinery session

The worker and refinery entries name agent-facing contracts at the execution and integration boundaries; they are not names for the runner and AME daemons themselves.

2.1.1. Bootstrapping a role with `yesod prime`

Yesod turns a general agent session into one of these factory roles through the `yesod prime` command. Priming is intentionally simple: the command renders a Markdown operating guide to standard output so that it can be placed into the agent's context at startup. It does not train a model, change its weights, or create durable memory. It gives a fresh session the factory's current operating doctrine.

Every invocation begins with a shared guide, `PRIME_GUIDE`. That common layer explains what Yesod is, separates the knowledge registry from the software factory, establishes Beads routing rules, introduces the team and authority model, and lists the essential commands. A role flag then appends the specialized guide for one seat. For example:

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

The first command combines the common guide, the planner's mission and workflow, and the peer-mail protocol. The second adds both peer communication and the separate human-authority escalation channel to the infra-monitor contract. These communication guides are **addenda**, not roles, so they can be composed with whichever primary role needs them. The refinery variant also appends a live rendering of its persisted merge queue, giving that session current queue state as well as doctrine.

The role-specific portion supplies the details that should not burden every agent: entry conditions, required checks, forbidden actions, handoff format, recovery rules, and role-specific commands. The task prompt can then stay narrow: it carries the task and its grounding facts, while the primer carries the role. This reduces repeated instructions and makes the same contract reusable across many short-lived sessions.

Priming is replayable by design. After context compaction, a cleared conversation, or a newly spawned replacement, the agent can run the same variant again to restore the role contract. The role's operational state lives separately in its `factory:<role>` topic. Before compaction or handoff, the agent records a checkpoint containing its address, focus, active note and Bead IDs, completed facts, decisions, blockers, and exact next action. A replacement re-primes, reads the newest matching checkpoint, and resumes from durable state rather than reconstructing the work from chat.

Automatic `factory:seat-state:<role>` notes serve a narrower purpose. They record launch metadata such as the address, runtime, model, and spawn parameters. They do not replace the operational checkpoint. A respawned singleton reads both: seat-state explains how the seat was launched; `factory:<role>` explains what the seat was doing. Mail remains a coordination channel, not the sole copy of a handoff.

Durable continuity therefore remains outside the prompt—in topic notes, work notes, Beads, PostgreSQL, Git, and queues—while `yesod prime` restores the instructions for working with that state. The roster is broader than the early five-role story told in the June blog; that story is useful history, not a current role count.

> **Terminology Trap — A primer is not a credential.** `yesod prime --mayor` can print the mayor guide for any caller. Priming establishes expectations inside an agent's context; authority still comes from the human channel, audited state changes, and deterministic service boundaries.

The control plane is not a chain of command based on claimed identities. Peer messages are lateral coordination. The `from` address on an agent message is self-asserted.

> **Invariant — Authority.** Human authority flows only through the designated human channel. Peer mail may carry evidence, requests, or advice; it does not carry Stephen's authority.

That rule prevents a compromised or confused agent from turning a plausible message into a production command. Deterministic services such as the merge controller operate within authority encoded in code and configuration. Agents can prepare exact recovery steps, but destructive or production actions outside those standing contracts still require the appropriate authority path.

2.1.2. The planning–dispatch boundary

The current contract makes the role boundary explicit. A planner grounds scope, chooses merge boundaries, constructs each feature's Bead tree, and records dependencies. It leaves every planned note unarmed. The dispatcher evaluates the completed plan, chooses a lane, records the routing reason, and arms the feature note.

This is more than division of labor. It prevents planning from quietly turning a checklist item into a separately routed feature, and it gives model choice an independent review point. The audited write path still matters regardless of the surface used, but the role owner is no longer ambiguous: planners structure; dispatchers schedule.

Likewise, no single transport owns arming. The web endpoint, `yesod codefactory-dispatch arm`, hold and unhold operations, note creation, and process instantiation (called **pouring**) can all initialize or change dispatch state through catalog-writing paths. The durable audit matters more than the surface used to reach it.

2.2. The Execution Plane

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

A runner is a deterministic daemon. It discovers eligible work, claims one note atomically, creates an isolated worktree, materializes the prompt and attachments, launches the lane's command, enforces time and token budgets, polls for kill requests, and reconciles the result.

The coding process is an agent, but the process around it is not. That difference allows the factory to tolerate a worker that hangs, exits incorrectly, forgets to commit, consumes too many tokens, or reports success without a branch.

Successful worktrees are removed after the remote candidate branch is verified. Failed or unverified worktrees are retained for diagnosis. This is the opposite of the older blanket claim that every worktree is kept forever.

2.3. The Integration Plane

The integration plane is the merge controller, the deterministic gate, and—for Yesod itself—the promotion transaction.

The controller is intentionally small and deterministic. Each tick it reconciles durable state, claims one lane at a time under a lease, and advances each merge attempt by a single proven

step—prepare a candidate, gate it, push it—recording every transition. It exercises no judgment about *whether* to merge; when a merge needs judgment, such as a conflict, it reroutes the problem to a bounded coding job that must re-earn the gate. Authority stays in code; intelligence is borrowed and re-proven.

This design limits daemon memory and stale-code drift. It also means the durable tables—not a long-lived worker's in-memory queue—must carry continuity across crashes.

2.4. The Data Plane

Three stores cooperate without pretending to be one database:

- **PostgreSQL** holds the catalog and operational ledger: tools, notes, statuses, lane changes, runs, runner heartbeats, mail, merge queue, merge jobs, merge summaries, gate runs, service status, and other telemetry.
- **Dolt-backed Beads** holds per-tool work graphs, parent/child structure, and dependency readiness.
- **Git** holds source history, candidate branches, `origin/main`, and the actual ancestry used to prove landing.

Each answers a different question. When operators try to make one store answer all three, confusion follows. A closed bead does not prove the code landed. An `awaiting_merge` note does not prove a queue row exists. A queue row marked blocked does not change the note's lifecycle state automatically. A deploy log does not change the merged SHA.

2.5. Why Determinism Matters

Agentic systems are good at semantic tasks: interpreting requests, reading unfamiliar source, selecting a strategy, and forming hypotheses. They are poor choices for repetitive enforcement:

- taking a row lock;
- comparing SHAs;
- checking a timeout;
- counting retries;
- verifying a remote ref;
- deciding whether two file sets overlap;
- refusing a launch above a hard cap.

Those operations should be code. Yesod's maturity comes less from adding roles than from repeatedly moving invariant enforcement out of prompts and into services.

> **Design Trade-off — More mechanism, less improvisation.** Every mechanical guard adds code and operational surface. It also makes a failure reproducible, testable, and visible without asking an LLM to remember the rule at the worst possible moment.

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

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, but they are not interchangeable.

3.1. Notes: Intent and Dispatch History

A work note is a PostgreSQL row with a stable identifier such as `ys-yes-ab12`. In the software factory, one feature-request or bug-report note represents one independently mergeable feature. Its important fields include:

- tool and kind;
- narrative body;
- lifecycle status;
- linked bead ID;
- `agent_dispatch` lane or the `hold` sentinel;
- acceptance and TUI smoke checks;
- retry, host-affinity, and verification metadata;
- audit history for status and dispatch changes.

Feature requests and bug reports use the implementation lifecycle. Usage notes use a smaller knowledge lifecycle. A planner temporarily moves an `open` work note to `planning` while grounding it, then returns the completed, still-unarmed plan to `open`. Dispatch and execution continue through `claimed`, `in_progress`, `awaiting_verification` or `awaiting_merge`, and finally `done`. `wontfix` and `superseded` are side dispositions.

The declaration is broader than the active automated path. In the pinned code, normal integration leaves a note at `awaiting_merge` throughout gating. A green merge moves it directly to `done`; a red gate changes the queue state while the note remains `awaiting_merge`. The declared `merging` status is therefore not a normal automated transition. `blocked` presents the inverse discrepancy: the database and runner logic use it, but it remains absent from `note_status.py`'s canonical lifecycle tuple.

That is why this book diagrams actual drivers rather than copying an enum.

> **Terminology Trap — Armed is not a status.** An `open` note with a real lane is armed, but it is not automatically claimable. The retry delay, dependencies, host routing, workspace readiness, and runner capacity must also permit a claim. An `open` note without a lane is invisible to ordinary runner selection. A held note stores `agent_dispatch = 'hold'` and is intentionally excluded.

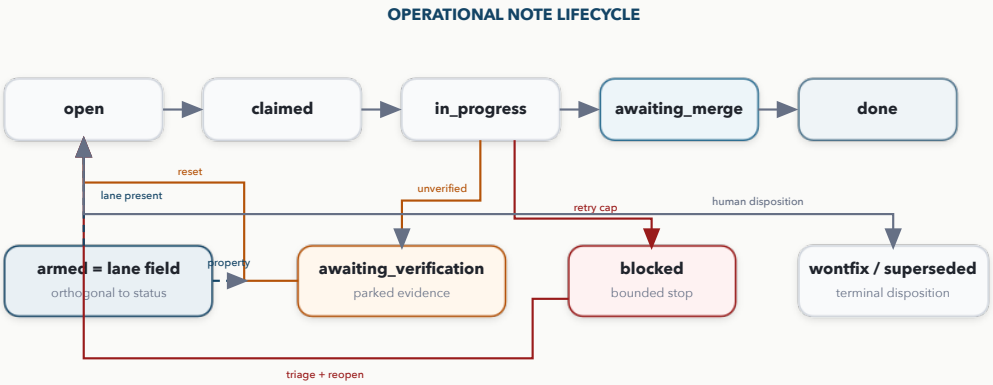


Figure 3.1.: The operational note lifecycle. Arming is shown separately because it is not a status.

3.2. Beads: Work and Dependency Structure

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. Dolt provides a versioned SQL backend for those workspaces.

The factory routes Beads operations through Yesod so the correct workspace and connection context are applied:

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

Bead dependencies order work **inside one feature tree**. They answer questions such as "must the query exist before the CLI wrapper is written?" or "do the tests depend on both implementation steps?" Their job is to keep a single feature on track, not to create a new branch or merge boundary.

3.3. Two Dependency Layers

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>
```

A note dependency is a merge-aware barrier. The downstream feature remains blocked while a prerequisite is merely `in_progress`, `awaiting_merge`, or `merging`. It releases only when every prerequisite is `done` or deliberately `superseded`. For example, an API feature and a front-end feature may both depend on a database feature, while the database feature's own migration and test Beads use ordinary Bead dependencies.

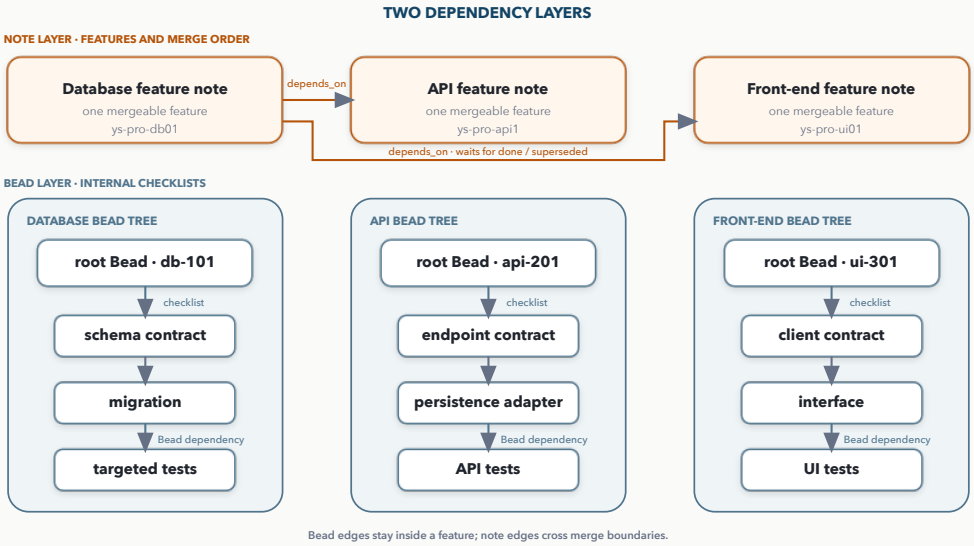


Figure 3.2.: Note dependencies order mergeable features; Bead dependencies order checklist work inside each feature.

The boundary rule is simple: 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 not merely a child. Promote it to a separate feature note with its own root Bead, then connect the feature notes with `depends_on`. Ordinary checklist Beads do not get child notes.

> **Invariant — Dependencies gate on integration.** Downstream features do not proceed merely because an upstream agent finished. The prerequisite must land or be deliberately superseded.

3.4. Lanes, Hosts, and Affinity

A **lane** is a named execution target such as an opencode-backed model or a native Claude CLI target. The lane registry exposes those targets to dispatch, and each runner advertises the lanes it can serve. A lane is not a machine.

The runner also respects:

- explicit host pins;
- soft host affinity that ages out;
- per-model host plans;

- tool workspace availability;
- note-level dependencies;
- retry backoff;
- the `hold` sentinel.

This distinction solved an early identity problem. A host should never pretend to have another runner ID merely to attract a task. Host preference belongs in routing data, not identity masquerade.

3.5. Three State Machines, Not One

The two currencies do not collapse the factory into two status fields. When operators ask, "What state is this work in?", the answer may require three coordination lookups:

1. **Note lifecycle** — the active path through `open`, `claimed`, `in_progress`, `awaiting_verification`, `awaiting_merge`, and `done`, plus side dispositions.
2. **Bead graph** — open or closed, ready or blocked by dependencies.
3. **Merge attempt** — the v2 integration state machine, immutable per try: `requested → claimed → preparing → gating → gate_green → pushing → succeeded`, with terminal states (`gate_red`, `push_raced`, `needs_rework`, …) that never reopen. The legacy `merge_queue` (`queued`, `gating`, `gate_red`, `held`) still exists, but only as an intake the controller bridges into attempts; authority lives in the attempt rows and their per-lane leases, not the queue statuses.

Those views can differ without making the system incoherent. Some differences are legitimate; others are repairable gaps between stores that cannot share one transaction. A note can remain `awaiting_merge` while its queue row is `gate_red`. A note can be `awaiting_merge` while queue enrollment is temporarily missing. A bead can be closed while gate-on-merge keeps a dependent from claiming. The purpose of reconciliation is not to force every store into the same vocabulary. It is to repair the relationships that should hold between them.

> **Operator Check — Ask the right store.** Use note status to understand lifecycle, Beads to understand readiness, the run ledger to understand execution, the merge queue to understand integration, and Git ancestry to understand what actually landed.

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

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 miss once runners and MicroVMs enter the story. It is also what makes Yesod different from a job queue with an LLM attached.

4.1. Tools as Durable Context

A registered tool has a name, ownership kind, description, repository or URL, binary registrations, and a stream of timestamped notes. The kinds distinguish tools Stephen owns (`internal`), tools built by colleagues (`team`), and public or third-party tools (`external`). The distinction informs search and recommendations: all else equal, `yesod ask` prefers an internal tool, then a team tool, then an external tool, while still allowing a better-fit external tool to win.

The basic operations are intentionally 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
```

The tool is not an active agent. It does not wake up and write a complaint by itself. But agents and workflows acting in a tool's context can file notes against any other registered tool. That makes integration friction durable.

Suppose a video publisher discovers that a PDF renderer emits filenames the upload tool cannot accept. The workflow can record a bug against the renderer, a feature request against the uploader, or a cross-cutting topic note about filename contracts. The problem stops living in a transcript between two tools. It becomes searchable catalog state with an ID, status, comments, attachments, and—if it becomes implementation work—a bead and dispatch path.

This is how tools "learn about each other" in Yesod: not by sharing mutable prompts, but by accumulating **addressable, attributable facts** in a common registry.

4.2. Querying One Tool in Its Ecosystem

`yesod query TOOL -q "..."` does more than send the tool's description to a model. Its context loader assembles:

1. the tool's metadata;
2. its timestamped feature requests, usage notes, and bug reports;
3. relevant snapshots of its Beads workspaces;
4. every registered process that references the tool, including the other steps.

The process context is crucial. A tool rarely has one meaning in isolation. A clip generator may be upstream of a captioner in one process and downstream of a transcript service in another. By showing the entire process, Yesod lets the model answer not only "what does this binary do?" but "where does it fit, what relies on it, and which operational notes change the recommended sequence?"

`yesod ask` works across the catalog. It can answer questions such as:

- Which tool do I already have for this job?
- What is the preferred way to publish an episode?
- Which registered process already contains most of these steps?
- Which tool has an unresolved bug that makes this workflow risky?

The query prompt instructs the model to cite note IDs and process step numbers. That turns an LLM response into a navigable explanation rather than an ungrounded recommendation.

> **Design Trade-off — Retrieval over omniscience.** Yesod does not ask a model to remember a private tool ecosystem. It assembles the current catalog, active notes, and process relationships at query time. The model reasons over evidence the user can inspect.

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 native directed acyclic graphs.

```
yesod processes add release-booklet \
  --description "Render, inspect, and publish a technical 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 DAG authoring. Independent steps become parallel roots or branches; later steps can depend on one or several predecessors. A step can also be a human handoff rather than a tool action.

The word **molecule** is not decorative. Under Architecture Decision Record (ADR) 005, the source of truth for a process is a reusable Beads prototype in a dedicated `proc` workspace.

The PostgreSQL `processes` and `process_steps` tables mirror that graph for catalog queries and the UI. That gives the process system two useful forms:

- a readable catalog description for search and composition;
- an executable dependency structure that can be instantiated, or **poured**, into a concrete run.

The authoring CLI keeps these forms aligned. It updates the catalog projection, writes or refreshes the prototype when the `proc` workspace is available, and appends version history. The `sync` command repairs a missed prototype write and migrates older table-defined processes.

The lifecycle has a small command vocabulary:

- `yesod processes validate` checks tool references and structural integrity;
- `yesod processes sync` repairs or migrates prototypes in the process Beads workspace;
- `yesod processes pour PROCESS` instantiates a process with `bd mol pour` and creates run notes;
- `yesod processes runs` lists concrete pours and their provenance;
- `yesod processes squash RUN` compacts a completed molecule into a durable digest;
- `yesod processes versions PROCESS` exposes append-only process history;
- the `yesod schedules` commands register recurring pours and evaluate when they are due.

The factory can therefore dispatch a feature, but it can also execute a repeatable business or production workflow whose steps involve multiple tools and people.

4.4. Composition Creates a Feedback Network

Once tools and processes share a registry, several feedback loops become possible.

A process teaches tools about dependencies. Querying a tool reveals the flows that depend on it. A breaking change can be assessed against actual process usage rather than a generic API description.

Tools teach processes about reality. Usage notes can record the command that works, a rate limit, a required ordering, or a known incompatible version. Catalog queries can surface drift from those facts before or during a run.

Runs teach the catalog about behavior. `yesod call TOOL ...` logs an invocation with command, result, timing, session, and the full captured stdout and stderr (bounding to tails, if added, would be a display-time concern). Telemetry can mine repeated failures into draft bug reports or process candidates.

Failures become cross-tool work. An agent running process step four can file a bug against the tool from step two, attach evidence, and link the note into the same durable work system used by the software factory.

Knowledge compounds. A stable fact discovered during one run becomes a usage note. A new tool can query that fact later without replaying the old transcript.

This is a lightweight knowledge graph even though it is implemented with ordinary relational joins and Beads edges. Tool-to-process references, note dependencies, topics, calls, and work graphs provide enough structure for useful ecosystem reasoning.

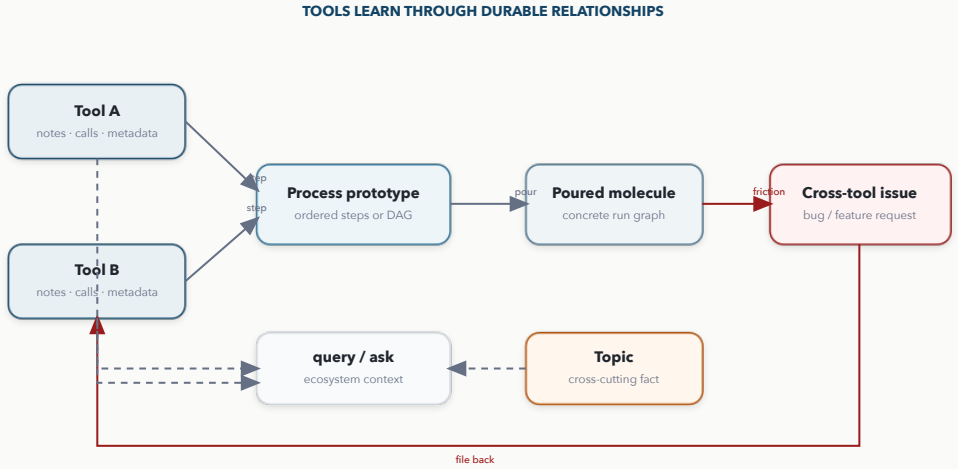


Figure 4.1.: Tools, processes, concrete runs, and cross-tool feedback form a learning loop.

4.5. Topics: Knowledge Without a Tool Owner

Not every durable fact belongs to one tool. Yesod topics hold cross-cutting knowledge such as factory authority, gate operations, or documentation standards. They have their own metadata and note lifecycle.

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.

That rule prevents the registry from degenerating into one giant notes table whose entries are impossible to retrieve correctly.

4.6. More Than a Factory Front End

The original registry remains useful even if no coding agent is running. It can answer how a local ecosystem works, preserve operational memory, compare tools, expose unresolved needs, and compose repeatable processes. The software factory is one consumer of that substrate: the consumer 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. Yesod preserves those transitions instead of forcing every observation into a ticket.

Together, these layers explain why the diagnostic 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 now returns to that request and follows each boundary in detail.

Part II.

From Intent to Merge

5 Planning and Dispatch

Chapter 1 followed a request for a command that explains why an armed note is not running. At first, that request is only an intention. It names a useful outcome, but not the code boundaries, dependencies, or proof that would let a worker execute it safely.

Planning closes that gap. It converts a human-shaped request into an execution contract that both agents and deterministic services can inspect.

5.1. Capture Before Decomposition

The **mayor**, Yesod's human-facing lead role, first preserves the request in the appropriate catalog form: a feature request for new behavior, a bug report for a defect, or a usage note for knowledge that may never become implementation work. The diagnostic command begins as a feature request. At this stage the note should state the need, the surrounding evidence, and the desired outcome. It should not prescribe an implementation that nobody has yet grounded in the source.

Planners are AI agents, not deterministic scheduler processes. Operationally, a planner is an `opencode` session primed with the planner role and communicating with the mayor through `yesod mail`. The mayor sends planning requests and context; the planner returns questions, findings, and proposed work graphs through the same mailbox.

Planning has two entry paths. Yesod maintains a standing **general planner**—a long-lived, role-primed `opencode` session—for ordinary planning work. The mayor may instead start a **task-specific planner** when a request benefits from explicitly chosen context, expertise, or attention. In either case, the planner reads the target repository and turns durable intent into a work graph. The selection method differs; the planning contract does not.

The planner first chooses feature boundaries. Each independently mergeable feature gets one note and one linked root Bead. Beneath that root, two to five child Beads form the feature's internal checklist and sequencing structure. Ordinary checklist children do not get their own notes, because they are not independent dispatch or merge units.

While grounding a feature, the planner sets its note to `planning`. When the source-grounded plan, scope fences, acceptance checks, and Bead descriptions are complete, the planner returns the note to `open` and leaves it unarmed. Lane choice and arming belong to the dispatcher.

For the diagnostic command, the planner might create three children:

1. add a query that computes the reasons a runner skips a note;
2. expose those reasons through a read-only CLI command;

3. add targeted tests and operator documentation.

The CLI depends on the query; the tests and documentation depend on both. An acceptance contract might require the targeted test plus a read-only invocation over fixture data. That structure gives later machinery something more precise than "make the command work": a dependency order, a bounded scope, and an observable result.

The plan should also say what **not** to change. Scope fences matter especially for LLM workers. A plausible adjacent refactor can consume the entire run budget, enlarge the review surface, and make the requested behavior harder—not easier—to prove.

5.2. Complexity Is Routing Data

Complexity is neither praise nor a judgment about code quality. It is evidence for choosing an execution lane.

Useful dimensions include:

- number of files and degree of coupling;
- database or migration changes;
- concurrency and lifecycle logic;
- unfamiliar external APIs;
- ambiguity in the requested behavior;
- blast radius if the change is wrong;
- whether the task modifies dispatch or merge machinery itself.

The diagnostic query might be locally small yet operationally sensitive because it interprets several state machines. That combination may justify a stronger reasoning lane even if the final diff is short.

Yesod records model pricing, run outcomes, and the reason behind dispatch changes. Over time, those records can replace a static `easy`, `med`, `advanced` table with an empirical routing policy: which lane completes this kind of work reliably, and at what cost? Chapter 13 develops that feedback loop.

5.3. Choose Merge Boundaries Before Task Order

The planner makes two different dependency decisions.

First, **Bead-level dependencies** sequence work inside one feature. In the diagnostic-command example, the CLI Bead depends on the query Bead, while the test-and-documentation Bead depends on both. These edges keep one implementation on track.

Second, **note-level dependencies** order features across merge boundaries. Imagine a larger product plan with three features:

- a database feature creates the durable schema;
- an API feature reads and writes that schema;

- a front-end feature consumes the API and schema-backed behavior.

The API and front-end are separate notes with separate root Beads. Both can depend on the database note:

```
yesod tool product note-depends ys-pro-api1 --on ys-pro-db01
yesod tool product note-depends ys-pro-ui01 --on ys-pro-db01
```

Those dependents do not release when the database agent finishes or when its branch enters the merge queue. They release only after the prerequisite note reaches `done` or `superseded`. This is how the planner says "get the database feature first."

As the dependency-layer diagram in Chapter 3 shows, the arrows between features and the arrows inside a feature have deliberately different meanings. If a proposed child needs a separate lane, branch, or merge decision, the planner promotes it to a feature note with its own root Bead. It does not attach a note to an ordinary checklist child or draw a cross-tree Bead edge and hope that it behaves like a merge barrier.

5.4. Pre-flight Questions

Before the dispatcher arms a planned feature, the control plane should be able to answer:

- Is there exactly one root Bead linked to the feature note?
- Do its child Beads describe only work that should land with this feature?
- Does the target tool have a routable workspace and repository?
- Is the required lane served by at least one runner?
- Do Bead dependencies express only within-feature ordering?
- Are all note-level prerequisite features `done` or `superseded`?
- Is there a test or acceptance strategy?
- Does the work require production authority that a coding runner must not have?
- Is the tool covered by an owned refinery profile or an upstream-contribution path?

These checks are not ceremony. Each one moves a predictable failure out of expensive agent runtime and into a cheaper planning correction.

5.5. Arming

The **dispatcher** arms a planned feature note by assigning a real execution lane to its `agent_dispatch` field. Arming can happen through the web API or the CLI, and the catalog records both the lane and the reason for choosing it. The planner never performs this step.

```
yesod codefactory-dispatch arm NOTE_ID --target TARGET --reason "..."
```

The reason matters as much as the target. "Advanced lane because the query crosses note, Beads, and runner state" is useful evidence for later quality and cost analysis; an unexplained lane assignment is not.

Arming does not claim the work, launch an agent, or change the note's lifecycle status. It only makes an otherwise eligible feature note visible to runners that serve the selected lane. The Bead children remain the worker's checklist; they are not armed separately.

When the work must not run, the control plane can park it explicitly:

```
yesod tool <tool> note-hold NOTE_ID --reason "requires operator access"
```

`note-hold` sets the note's **disposition** to `held`, removing it from ordinary claiming without pretending that implementation is complete or that code failure has blocked it. (This `held` disposition is a distinct axis from the `agent_dispatch = 'hold'` sentinel that suppresses individual child notes during root dispatch; both keep work from running, by different means.)

At this point the request has crossed from planning into execution eligibility. The control plane has supplied structure and routing judgment; the next stage is mechanical. A runner must decide whether the armed work is eligible **now**, claim it exactly once, and govern the attempt.

> **Operator Check — Armed but not running.** Check, in order: note status and lane; hold state; linked root Bead and checklist readiness; note dependencies; runner lane coverage; host pin and affinity; retry delay; repository and workspace staging; and runner capacity.

6 The Runner

An armed note has permission to compete for execution; it does not yet have an owner. The **runner** is the deterministic daemon that turns eligibility into one governed attempt.

Its service is named `codefactory-dispatch`, which creates a terminology trap. The runner daemon is not the dispatcher agent. An agent uses judgment to choose and arm a lane; the daemon repeatedly evaluates rules, claims work, and launches the selected lane's command.

6.1. The Ten-Second Loop

By default, each runner polls every ten seconds. A cycle has three jobs: recover stale ownership, discover eligible work, and claim no more work than available capacity permits.

Recovery comes first. The runner can reclaim work from a peer whose heartbeat is older than 300 seconds, but it also consults the run ledger before declaring an attempt dead. That second check matters: a live worker should not lose ownership merely because load delayed its daemon heartbeat. On a slower cadence, the loop also reaps stalled notes while leaving deliberately parked `awaiting_verification` notes alone.

Discovery then intersects two durable inputs:

1. armed candidate notes in PostgreSQL;
2. dependency-ready beads in the relevant per-tool workspaces.

A fast path intersects the small armed set with the ready set before performing expensive per-bead subprocess checks. This is both an optimization and a statement of architecture: neither an armed note nor a ready bead is sufficient by itself.

For a note to remain eligible, it must satisfy all of the applicable rules:

- its lifecycle status is `open`;
- it names a real lane served by this runner;
- it is neither held nor superseded;
- its retry delay has elapsed;
- its note dependencies are satisfied;
- its host pin or aged soft affinity permits this host;
- it is the armed feature note linked to the root governing a dependency-ready checklist child;
- no closed blocker has a linked note that still awaits integration;
- guards for terminal beads and workspace routing pass.

For the diagnostic-command example, this is the moment at which planning becomes executable fact: the root note is open and armed, its first child is ready, this runner serves the selected lane, and no dependency or retry delay says "not yet."

6.2. The Claim Boundary

Several runners may discover the same candidate. Only one may own it.

The runner claims the note inside a PostgreSQL transaction using `SELECT ... FOR UPDATE`. It rechecks status and capacity while holding the row lock, records runner ownership, and increments the runner's task count before releasing that lock. Only PostgreSQL deadlocks receive automatic retries, with short exponential delays.

Bead assignment happens **after** the database claim. If the assignment fails, the runner releases the catalog claim instead of leaving a half-owned task.

This sequence is not a distributed transaction across PostgreSQL and Dolt. Yesod cannot atomically commit ownership in both systems, so it uses a short authoritative claim followed by an explicit repair path. The design does not eliminate partial failure; it gives partial failure a detectable shape and a deterministic response.

6.3. Worktree Isolation

Once claimed, the runner fetches the repository and creates an isolated Git worktree based on current `origin/main`, normally beneath `/tmp/yesod-workdir`. The feature receives one candidate branch. Its ready checklist children execute in governed worktrees while successful child commits accumulate on that branch. Note attachments are materialized into the relevant worktree.

The runner still contains compatibility paths for older exact child-note records. They explain historical state and can help recover legacy work, but the planner no longer creates that shape. Current plans promote independently routed or merged work to a separate feature note and root Bead.

The isolation protects concurrent workers from one another, but only if the worker respects it. The worker canon therefore forbids switching branches, rebasing, merging, or resetting away from the prepared task branch. It also requires incremental commits because the process can be killed at any time. A committed partial layer can be inspected or recovered; an uncommitted transcript of good intentions cannot.

For multi-repository work, the runner adds per-tool repository paths, Beads workspaces, and optional deploy keys. Identity remains host-local: routing may prefer a host, but a runner never adopts another runner's identity merely to attract work.

6.4. Timers That Define Failure

Agentic execution can hang without crashing. Yesod therefore treats time limits as part of the execution contract, not as operator folklore.

Control	Default
Runner poll	10 s
Runner heartbeat	30 s
Dead-runner threshold	300 s
Stalled-note threshold	300 s
Worker runtime cap	1,800 s
Governance kill poll	5 s
Token sample	30 s
SIGTERM-to-SIGKILL grace	30 s
Acceptance check	120 s
TUI smoke check	600 s
Max execution retries	3
Soft affinity age-out	300 s

Together, these defaults answer operational questions precisely: how often a runner should appear alive, when ownership becomes suspect, how long a worker may run, and when preference for one host must yield to fleet availability. Without such thresholds, "stuck" means only that an operator has become impatient.

6.5. The Run Ledger

Every governed attempt opens a durable run row. The ledger records the note and bead, runner and host, lane and model, timestamps, status, exit code, worktree, result commit, token and cost data, verification results, outcome verdict, failure classification, and log archive details.

Rows left `running` by a dead daemon can later be reconciled to `lost`. Cross-host kill requests are durable fields that the owning runner polls. The ledger therefore answers a different question from the note:

- the note describes the lifecycle of the requested change;
- the run ledger describes each attempt that spent time and money on it.

> **Invariant — Attempts are not erased by success.** A note that lands on its third run still incurred the cost of all three runs. Quality and cost analysis must include failures, timeouts, and no-progress attempts.

With the claim recorded and the worktree prepared, the runner can finally start the coding agent. The next boundary governs what that agent may do—and what evidence must exist before its work is allowed to proceed.

7 Governed Agent Execution

The runner now hands the prepared worktree to a coding agent. This is the most open-ended stage of the pipeline: the worker must understand unfamiliar code, choose an implementation, and respond to evidence from tests. Yesod gives it substantial freedom—but only inside a narrow envelope.

The worker may inspect source and history, edit files, add targeted tests, and commit. It may not decide that a missing branch is "close enough," extend its own deadline, or merge to `main`. Judgment belongs inside the process; authority remains outside it.

7.1. The Worker Canon

A dispatched worker does not begin with an improvised job description. Its task prompt, the `yesod prime --worker` guide, the repository's `AGENTS.md`, and the bundled skill all include a generated **worker canon**: a common operating contract for agent behavior at this boundary. Generating that contract once avoids hand-copying safety-critical instructions across several surfaces and reduces drift between them.

The canon's main disciplines are:

- route Beads commands through Yesod;
- inspect current state before editing;
- work through children in dependency order;
- commit incremental, recoverable layers;
- run targeted tests rather than the repository-wide merge gate;
- report progress through `yesod dispatch-agent-log`;
- remain on the prepared task branch;
- never close the root bead;
- return durable tool knowledge to Yesod.

The last rule matters beyond the current change. If the worker discovers a stable command, architectural constraint, or recurring trap, that knowledge should outlive its context window. Recording it as tool knowledge makes the next worker—and any process composed around that tool—better informed.

For the diagnostic command, the canon keeps the worker focused on the planned query, CLI surface, and proof. It does not prevent the agent from choosing a good implementation; it prevents the implementation session from quietly becoming a merge operator, release manager, or unbounded refactoring project.

7.2. Governance Around the Process

The runner launches the lane command in its own operating-system process session and places a deterministic governance loop around the entire process tree. That loop watches:

- the wall-clock deadline;
- the token budget;
- durable cross-host kill requests;
- process exit;
- the termination grace period.

When a limit is crossed, the runner sends SIGTERM and, after the configured grace period, SIGKILL to the process tree. The agent cannot negotiate with the deadline in prose, and governance is not limited to watching the parent process.

Progress events solve a different problem. They tell operators what the agent believes it is doing and provide liveness while ordinary output may be buffered. They are not proof of delivery. A run can narrate beautifully and still produce no commit.

7.3. What Counts as Success

When the worker exits, the runner evaluates artifacts rather than confidence.

For an ordinary code task, the branch must contain real commits beyond the recorded base SHA. Declared acceptance commands and terminal user interface (TUI) smoke checks contribute additional evidence with deliberately precise semantics:

- an acceptance result of `fail` blocks the run;
- a TUI smoke result of `fail` blocks the run;
- a missing or skipped optional check is not treated as an explicit failure;
- a `task_kind=data` task follows a different contract: its acceptance probe replaces the commit gate and must pass.

This is slightly less absolute than older documentation that said every declared check must return `pass` for every code task. The source implementation—not the older slogan—is authoritative.

If the local evidence is sufficient, the runner force-pushes the prepared candidate branch and verifies that the remote ref resolves. A local commit alone is not enough: the next service runs in another checkout and needs a durable branch it can fetch.

For a feature-note dispatch, normal success advances the note to `awaiting_merge` and then attempts to enroll it in the merge queue. Those are two durable operations rather than one transaction. If a failure leaves a gap between them, refinery reconciliation can reconstruct the missing queue entry from authoritative state. The note and its root Bead remain the merge unit; completed checklist children are evidence inside that unit, not independently enrolled features.

Successful worktrees and local branches are removed after the remote candidate is verified. Failed and unverified worktrees remain available for triage because they may contain evidence or useful partial work that never became a valid merge candidate.

> **Invariant — A clean exit is not delivery.** Ordinary code work requires commits and a verified remote branch; a data task requires a passing acceptance probe. Every task still needs its contract-appropriate evidence and a downstream integration result. Exit zero alone is never the definition of success.

7.4. Failure Has Two Axes

The runner records failure along two independent axes: **what happened to the work** and **why the attempt ended**.

Outcome verdicts describe the work product:

- work complete but missed by the ordinary success path;
- useful work present but uncommitted;
- partial progress;
- no progress;
- infrastructure failure.

Failure classifications describe the attempt's behavior or stopping condition, including fast-fail, runaway, no-op, gate hang, lost work, timeout, token budget, killed, and missing logs.

The distinction is operationally important. "No commit" can mean the model did nothing, a required tool failed before launch, useful edits remain uncommitted, or the requested behavior was already present. A retry policy based only on exit code would treat those cases alike, waste money, and sometimes destroy the best evidence.

7.5. Bounded Retries

Each ordinary failed attempt increments the note's retry count; the default cap blocks the note when that count reaches three. Backoff between eligible attempts grows exponentially in minutes. Failures attributed to model capability can escalate to a stronger lane, while rate limiting and infrastructure failures can avoid consuming the ordinary allowance. Two consecutive no-progress verdicts stop early and move the note to `blocked` rather than paying for a third evidence-free attempt.

The rule behind these details is simple:

> **Design Trade-off — Retry only when the next attempt is meaningfully different.** Backoff, a repaired dependency, a stronger lane, or a triaged fix can change the experiment. Repeating the same failing conditions is not resilience.

Successful execution ends with contract-appropriate evidence: normally a verified remote branch, or for a data task, a passing acceptance probe. That evidence is not a merge; it is the durable handoff from the execution plane to the integration system.

8 The Merge Controller

It turns out that one of the hardest operations in the whole factory is merging.

Everything before this point has been about *parallelism* — launch a dozen coding agents, give each its own worktree, let them work without stepping on one another. That is the easy half. The hard half is the other end: eventually, every one of those changes has to land in the *same* codebase, on the *same* `main`, and be true at the same time. Parallel work is cheap to start and expensive to reconcile.

Consider a few of the ways reconciliation goes wrong, all of which the factory hits routinely:

- Two agents, working in parallel, both edit the same file. Each branch is fine on its own; together they conflict.
- An agent branches off `main`, works for twenty minutes, and by the time it finishes, `main` has moved. Its branch is merge-clean but carries a test that pins behavior `main` has since changed — green in isolation, red against reality.
- A merge starts, the gate passes, and the process dies one step before the push. Did the change land or not? A second process has to be able to answer that from durable state, not a hunch.
- Two runners both believe they own the merge for the same repository, and both start pushing.

Any one of these, handled by guesswork, corrupts shared history — and shared history is the one thing in the factory you cannot cheaply un-corrupt. This is why merging earns its own chapter, and why Yesod recently **rewrote the entire merge engine** to get it right. The old engine was a standing agent that polled a queue and launched a one-shot worker to merge each batch; it worked, until the cases above stacked up. The replacement is a deterministic **merge controller**: it reads durable state, derives the single next action, takes it under a lease, and — when the evidence is missing or contradictory — refuses and says exactly why.

But *deterministic* here does not mean *never uses AI*. It means the controller itself never guesses. When a merge genuinely needs judgment — most often because two changes conflict at the same lines — the controller does not resolve the conflict with a heuristic and does not paper over it. It **reroutes the problem back into the factory** as a fresh, bounded coding job, and then treats whatever that agent produces as an untrusted proposal that has to re-prove itself through the full gate before it can become shared history. (We follow that path in detail below, in "When the Merge Itself Conflicts.") Judgment is used freely; *authority* is never delegated. Merging is precisely the operation where guessing is catastrophic and determinism about *authority* is cheap.

The rest of this chapter follows one change — our diagnostic note from Chapter 1, now sitting as a verified candidate branch — through the controller. First the way it is *supposed* to go,

then, one at a time, the ways it can fail and what the controller does about each. But first, the words.

8.1. The Vocabulary of a Merge

The controller's design is small, but it leans on a handful of terms that mean something precise here. It is worth pinning them down before we watch them work.

- **Lane.** The unit of merge serialization: the pair `(repository, target branch)` — the repository whose history is being changed, and the exact branch the change lands *on*. "Merging into the `yesod` repo's `main`" is one lane; "merging into `ttrac`'s `main`" is another. The rule that organizes everything else is *one writer per lane*: at most one merge may touch a given lane at a time, while different lanes proceed fully in parallel. A lane is not a queue and not a host — it is simply the identity a merge must hold exclusively to touch a branch.
- **Target branch — and it is not always `main`.** A lane's target is whatever branch the change is landing on, and while that is usually `main`, it need not be. A single note typically targets `main` directly. But a larger **feature** — one big enough to span *many* notes — can target a shared **feature branch** (an *integration branch*) instead: each contributing note merges into that feature branch as its own lane, the feature accumulates commit by commit as its notes land, and only when the feature is whole does the feature branch itself merge into `main` as one final lane. So `main` is the common target, not the only one; the lane model serializes merges onto *any* branch that several changes need to converge on, which is exactly what lets a multi-note feature be assembled safely off to the side before it touches shared history.
- **Candidate.** The exact merged tree being considered — the source branch merged onto an exact base commit — identified by its exact SHA. Not "roughly this branch": a specific, fetchable commit.
- **Attempt.** One try at integrating one note into one lane. An attempt is a durable row with a lifecycle, and — this is the load-bearing part — it is **immutable once it ends**. There is no "reopen"; a retry is a *new* attempt.
- **Lease.** A time-boxed claim on a lane, held by whichever controller process is currently working it: an owner, an expiry, and a **fencing token**.
- **Fencing token / generation.** A per-lane counter that only ever increases. Every write must present the current token; a write bearing an old one is refused. This is how a stalled owner is prevented from damaging a lane it has silently lost.
- **Admissions fence.** A standing pause on the whole merge intake, guarded by a capability token and a revision number. When it is closed, no new work is admitted. It is what lets a deployment (Chapter 10) freeze all merges for its duration.
- **Contract / commitment.** Throughout the merge system, a *contract* is a rule the code refuses to violate — "a terminal attempt never becomes active again," "one writer per lane," "the exact commit that passed the gate is the commit that is pushed." A *commitment* is the durable evidence that a contract was honored: a fencing generation, a lease row, a recorded candidate SHA. The controller's job is to turn contracts into commitments, one attempt at a time.

Hold onto **lane**, **attempt**, and **lease** especially. Almost everything the controller does is: claim a *lane* by taking a *lease*, run an *attempt*, and leave behind a *commitment* that says what happened.

8.2. The Good Case

Start with the happy path, because it is short.

Our diagnostic note has reached `awaiting_merge`: a worker produced a real commit on a candidate branch, pushed it, and enrolled the note in the merge queue (Chapter 7). The controller runs a loop — a *tick* — every few seconds. On the tick that notices our note, this is what happens.

The controller sees an eligible note for the `yesod/main` lane. It opens an **attempt** — a fresh row, state `requested` — and claims the lane by taking a **lease**: it writes itself as the owner, sets an expiry, and stamps the attempt with the lane's current **fencing token**. The attempt moves to `claimed`.

Now it does the work, one bounded step per tick, each step advancing the attempt's state and each step gated on the lease still being valid:

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

In `preparing`, it builds the **candidate**: it merges our branch onto the exact current `origin/main` and records the resulting tree's SHA. In `gating`, it hands that exact candidate SHA to the gate (the next chapter) and waits. The gate comes back green, so the attempt moves to `gate_green`. In `pushing`, the controller pushes the exact candidate SHA — the one that was gated, not "approximately this branch" — to `origin/main`, and confirms the remote now points at it. The attempt reaches `succeeded`, the note is marked `done`, and a durable merge record is written.

That is the entire happy path: one lease, one candidate, one gate, one push. Notice what is *not* here — no batch, no global lock, no negotiation. A note that merges cleanly barely troubles the controller. The design earns its keep entirely in the failure cases, which is where we go next.

8.3. When the Gate Says No

The first and most common failure: the gate runs and the tests fail.

The controller does *not* reset the attempt and try again. It **ends** the attempt at the terminal state `gate_red`, and parks the note as `needs_rework` with the per-shard failure evidence attached. If the note is later fixed and re-armed, a **new** attempt is created — a fresh row, linked to the old one by a `prior_attempt_id` pointer.

This is the immutability contract, and it looks like pedantry until you see what it forbids. Picture a mutable status field cycling `gating → gate_red → gating` on a single row. A slow or crashed worker, waking up late, could write a stale `gating` over a fresh `gate_red` and

revive a dead merge. Immutable attempts make that unrepresentable: a terminal row has *no* outgoing transitions, so no late writer can bring it back.

> **Invariant — A terminal attempt never silently becomes active again.** Forward progress is always a new, freshly-fenced attempt. Here, "retry" is a noun, not a verb you apply to an old row.

It also keeps the books honest. A note that merges on its third attempt cost three gate runs, and all three rows survive — which is exactly what the cost and reliability accounting in later chapters needs.

8.4. When the Owner Stalls

Now a subtler failure: the controller claims our attempt, starts gating, and then its host pauses — a long GC, a network partition, a VM that gets descheduled. The work is not dead, just frozen, and there is no reliable way to tell the two apart from outside.

This is what the **lease** and **fencing token** are for. The lease has an expiry. When it lapses, a successor process may reclaim the attempt — and when it does, it bumps the lane's fencing generation and takes the higher token. Later, the original owner wakes up and tries to push. Its token no longer matches the lane's current generation, so its write is refused *before it is even evaluated*. It cannot commit anything. It learns nothing that could tempt it to continue.

> **Design Trade-off — Fencing over coordination.** Yesod does not try to make the two owners agree on who is in charge. It makes the *stale* owner's writes impossible to commit. Coordination is a conversation you can lose; fencing is a fact you cannot argue with.

This is also the answer to "two runners both think they own the merge." They can both think so; only one holds the current fencing token, and only that one's push lands.

8.5. When `main` Moved Underneath

Our candidate was built by merging onto a specific `origin/main`. What if, between gating and pushing, some *other* lane's merge advanced `main`?

The controller checks, and if the base it built on is no longer the tip, the attempt ends at the terminal state `push_raced`. It does not force its now-stale candidate over the top of the newer history. A new attempt is created, which builds a *fresh* candidate against the new `main` and gates that. The contract — *the exact commit that passed the gate is the commit that is pushed* — is never bent; if reality changed, the controller re-derives and re-proves rather than push something that was never actually tested against the current tree.

This is also the mechanism behind a failure that looks like a code bug but isn't. A branch based on an older `main` can be perfectly merge-clean and still fail the gate, because it carries a test pinning behavior that `main` has since changed. That is not a flaky test; it is *semantic staleness*, and a deterministic candidate built against current `main` is exactly what surfaces it.

8.6. When Another Change Wants the Same Branch

Two notes both target `yesod/main`. In the old batch model they might have been merged together and gated as a unit, with an isolation pass to find the culprit when the batch went red.

The controller's answer is simpler: they are the same **lane**, so they *serialize*. One attempt holds the lane's lease; the other waits for it. There is no batch to bisect and no isolation pass, because the two changes were never combined into an untested interaction in the first place. Meanwhile, a note targeting `ttrac/main` — a *different* lane — merges concurrently, unaffected. Head-of-line blocking is avoided not by cleverly reordering a queue around blocked entries, but structurally: an unrelated change was never queued behind the blocked one to begin with.

The old engine's best observation still holds, though, and is worth keeping: *file conflicts reveal architecture*. When many logically-independent changes all serialize because they all touch one monolithic module, the lane model makes that contention visible. Faster gates reduce the sting; decomposition — not scheduler cleverness — removes it.

8.7. When the Merge Itself Conflicts

Serialization decides the *order* in which two changes to a lane land; it does not guarantee they land *cleanly*. Two notes can touch the same lines, and when the second one's candidate is built — merged onto the now-updated `main` that already contains the first — Git reports a conflict. The merge does not apply.

Here the "deterministic" controller does something that can look, at first, like a contradiction: **it calls in an AI.** But it does so on its own strict terms, and those terms are the whole point.

The attempt ends with a durable `merge_conflict` outcome, and the controller does *not* try to resolve the conflict itself — no heuristic, no three-way guess. Instead, on the next tick, the conflict-repair pass reroutes the whole problem back through the factory exactly as if it were new work. It opens a **bounded** conflict-repair job, arms the note to a coding lane — `codex-gpt-5.6-sol-xhigh`, falling back to `opencode-kimi-k3` — and a fresh agent is dispatched into a worktree with a narrow brief: reconcile the source branch with the snapshotted target, preserve the change's intent, resolve the recorded conflict, commit — and, as always, do not switch branches or force-push.

Then comes the part that keeps the system honest. Whatever that agent produces is **not** treated as a merge. It is submitted as a **fresh immutable attempt** and must pass the **complete gate again**, from scratch. The model is allowed to *propose* a resolution; it is never allowed to *authorize* one. If the repair earns a green gate, the note merges like any other. And because the repair job is *bounded*, not infinite, exhausted repairs do not loop forever: the controller stops guessing and opens an operator-inbox item asking a human to reconcile the branch by hand or supersede the note.

> **Invariant — AI proposes; the gate disposes.** Yesod does use a model to *resolve* merge conflicts, because that is genuinely a judgment task. It never lets the model's output *become* shared history without re-proving itself through the same gate every other change faces. Judgment is delegated; authority is not.

This one mechanism is the redesign's philosophy in miniature. The controller stays deliberately dumb, because being dumb about *authority* is what makes it safe. When intelligence is actually required, it neither fakes it nor bypasses its own contracts — it hands the problem to the factory it is part of, and waits for the result to earn its place through the gate.

8.8. When the Door Is Closed

Sometimes the right answer to "may this merge proceed?" is "not right now." When Yesod deploys *itself* (Chapter 10), it must freeze all merges for the duration, so that the code being promoted to production is a still target.

That freeze is the **admissions fence**. It is a standing, durable pause guarded by a capability token and a monotonic revision. While it is closed, an attempt that tries to enroll observes the pause and is refused — cleanly, with the current revision named. Admitting work is therefore a small, explicit ceremony: prove the set of eligible notes is the intended one, resume with the exact token and revision, let precisely the intended attempt come into being, then close the fence again with a fresh token and a new revision before the gate even starts.

This is the inversion the whole redesign is named for. Merging is not a default right the system occasionally withholds; it is a **granted privilege**, and the grant itself is a commitment — a revision number and a token *hash* written to the store, the token itself never displayed.

8.9. When the Process Simply Dies

Finally, the failure that underlies all the others: the controller process can die at any point — after any state transition, between any two steps.

The design's answer is that a tick keeps no authoritative state in memory. Everything that matters — the attempt's state, the lease, the fencing generation, the recorded candidate SHA — is a durable row. A process that dies mid-tick is recovered by the *next* process recomputing its position from those rows rather than from a cached phase. A crashed tick costs a tick, not a corruption.

> **Invariant — Recovery recomputes; it never remembers.** Restart-safety comes from re-observing the world — pointers, rows, leases — not from trusting what a previous process thought was true.

8.10. The Tick, Assembled

Put the failure handling together and the controller's loop is deliberately unglamorous. On each pass it:

1. runs the conflict-repair pass — arms bounded AI repair jobs for conflicted merges and folds their re-proven results back in (see "When the Merge Itself Conflicts") — *before* intake, so a prior tick's outcome is settled before the overlay re-derives it;
2. bridges the legacy merge queue into v2 attempts (the dispatcher still writes that queue; the controller is now its only consumer — the retired supervisor that used to drain it is gone);
3. runs the scheduler: renews the leases it owns, reclaims expired ones under a higher fencing token, claims new attempts up to a global capacity, and advances each owned attempt by exactly one step;
4. projects outstanding recovery work and routes anything genuinely ambiguous to an **operator inbox** — a deterministic queue of human decisions, never an agent asked to guess.

Every step is bounded and idempotent. There is no cleverness in the loop, and that is the point: the intelligence is in the *contracts* — one writer per lane, immutable attempts, exact-candidate pushes, fence-gated admission — and the loop merely turns them, tick by tick, into commitments.

The controller decides *whether* a candidate may join shared history and *when*. It never decides whether the candidate is *correct*. That judgment is delegated, every single time, to a gate that runs the tool's own tests against the exact candidate tree — and, for Yesod's most demanding path, runs them inside a disposable machine that can be thrown away the instant it has spoken. That gate is the next chapter.

9 The Deterministic Gate

The merge controller of the last chapter decides *whether* a candidate may join shared history and *when*. It never decides whether the candidate is any *good*. That single, absolute judgment — is this exact tree correct enough to become `main`? — is delegated, every time, to the **gate**.

A gate sounds simple: run the tests, read the exit code. It is not, for three reasons that the factory hits constantly.

- **A false green is unrecoverable.** If the gate says "pass" and it is wrong, a broken commit becomes shared history, and every change that follows builds on top of it. The controller trusts the gate *absolutely*; that trust has to be earned by the gate, not assumed.
- **The environment is a variable, not a constant.** Run the same tests on a laptop that is low on disk, or that has a stray daemon holding a port, or whose Postgres has yesterday's schema, and you get an answer about the *laptop*, not the code. A verdict is only meaningful if the thing that produced it is known and clean.
- **A test suite can hang, not just fail.** An agentic change can leave a process waiting on input forever. "Didn't pass" and "never answered" are different facts, and confusing them wastes money and stalls the lane.

So the gate is not "run pytest." It is a small contract about *what a verdict means, where it was produced, and what to do when no verdict exists at all*. This chapter walks a candidate through that contract — the way it is supposed to go, then the ways it deviates — but first, as before, the words.

9.1. The Vocabulary of a Gate

- **Gate.** The authority that evaluates one candidate tree and returns a verdict. For Yesod's own repository the gate is its test suite, run as `uv run pytest -q -p no:cacheprovider -n auto -m 'not bd_integration'` — the fast, pre-merge tier. (A slower `bd_integration` tier runs *after* merge through CI and can warn, but never blocks a merge.) Each tool brings its own gate command; the contract around it is the same.
- **Candidate SHA.** The exact merged commit being judged — never "the branch," always a specific, fetchable SHA. The gate checks out *that commit* and nothing else. This is the same candidate the controller built in `preparing`; the gate and the merge agree on one identity.
- **Verdict.** A structured, trustworthy statement that the candidate was *evaluated* and *judged* — green (it passed) or red (it failed). A verdict consumes a gate attempt and feeds the controller's decision.

- **Infrastructure failure.** The opposite of a verdict: the gate *could not run* — the machine never came up, the connection dropped, the request timed out. No judgment about the candidate exists. This distinction is the load-bearing idea of the whole chapter.
- **Gate execution.** One concrete run of a gate on one candidate, tracked as its own small state machine (`submitted → starting → running → {green, red, timed_out, cancelled, lost}`), so that a run which never reports back is a nameable state (`lost`), not a mystery.
- **Backend.** *Where* the gate runs. Yesod supports a **local** backend (a subprocess on the host) and a **remote MicroVM** backend (a disposable virtual machine in the cloud). Same contract, two executors.
- **Broker and provider.** The **broker** is the controller-side half that persists the intent to gate and reads back the result; the **provider** is the backend-side half that actually launches and observes a run. They meet at an **idempotency key**.
- **Idempotency key.** A deterministic fingerprint of "this gate, this candidate, this lane." It is how a crashed controller *recovers* a gate instead of *relaunching* it — ask the provider "what happened to the run with this key?" rather than starting a second one.
- **Loopback proxy, MicroVM, shard, fleet.** The remote backend's plumbing: a **MicroVM** is one disposable virtual machine that runs one gate; the **loopback proxy** is a host-local service (reachable only from `127.0.0.1`) that holds the cloud credentials and owns the MicroVMs' lifecycle; a **shard** is one slice of a test run when the suite is fanned out for speed; the **fleet** is all the MicroVMs currently alive.

Hold onto **verdict** versus **infrastructure failure** above all. Almost every deviation in this chapter is really one question: *did the gate actually judge the code, or did it just fail to run?*

9.2. The Good Case

Our candidate — the exact tree the controller built by merging the diagnostic branch onto current `main` — is ready to be judged. The controller's attempt is in `gating`.

The controller does not run the tests itself. It hands the **broker** a request naming the exact candidate SHA and the gate to use. The broker computes the **idempotency key**, writes down its *intent* to gate — before anything external happens — and submits once to the **provider**.

For Yesod's demanding path the provider is the MicroVM backend. It asks the **loopback proxy** for capacity; the proxy, holding the cloud credentials, starts a fresh **MicroVM**. Inside that VM a small server does something a laptop cannot promise: it builds the world from scratch — clones the repository, checks out the *exact* candidate SHA, stands up its own private PostgreSQL — then runs the tool's test command, fanned across **shards** for speed. When the suite finishes, it returns a **structured verdict**: green.

The broker records the green verdict against the idempotency key; the controller's attempt moves from `gating` to `gate_green`; and, because the executor was disposable, the run's telemetry has already been mirrored into shared storage, so operators can inspect what happened even after the VM is gone. The controller proceeds to push the *exact candidate SHA that was gated* — not a re-merge, not "approximately this branch."

That is the happy path: submit once, run against an exact commit in a clean disposable world,

return one trustworthy verdict. Everything else in this chapter is about what happens when the verdict is *red*, or when there is no verdict at all.

9.3. When the Tests Fail

The straightforward deviation: the gate runs and a test fails.

This is a **verdict** — red. The candidate was genuinely evaluated and genuinely rejected. It consumes a gate attempt, and the controller ends the merge attempt at `gate_red` and parks the note for rework, with the per-shard failure evidence attached so a human or agent can see *which* tests failed and why. There is nothing ambiguous to resolve; the system was asked a question and got an answer it can trust.

One flavor of red is worth naming because it looks like a bug and isn't: *semantic staleness*. A branch cut from an older `main` can be perfectly merge-clean and still fail, because it carries a test pinning behavior that `main` has since changed. That is not flakiness — it is the deterministic candidate doing its job, catching an interaction the branch never tested against current reality. The fix is a rebase-and-retry, not a retry-and-hope.

9.4. When the Gate Cannot Run

Now the deviation the whole contract exists for. The proxy is unreachable, or the MicroVM never boots, or the request times out. The tests never ran.

This is **not** a verdict. No trustworthy statement about the candidate exists, so the system must not pretend one does. The remote client tries the backend at most twice, invalidating any stale warm-pool state after the first failure; if the second attempt also fails to produce a run, it raises a distinct *infrastructure* error. Crucially, that error does **not** consume the merge attempt's gate budget and does **not** fall back to running the tests locally. The work simply stays queued until the gate can actually run.

That "no fallback" choice is deliberate and worth defending:

> **Design Trade-off — Visible stall over silent fallback.** A local fallback would keep merges flowing during an outage — but it would silently change the isolation, performance, and resource assumptions the verdict depends on. Yesod chooses a *visible stalled lane* over a green light produced under quietly different rules. A stall you can see is a bug report; a fallback you can't is a future incident.

Two properties make this safe rather than fragile. First, the boundary **fails closed**: a configured remote-gate URL *must* point at the local loopback proxy, and a direct cloud URL is refused outright — rejecting it surfaces as infrastructure trouble rather than quietly rerouting tests to the host. Second, recovery is **memoryless**: because the broker wrote its intent under an idempotency key *before* submitting, a controller that crashed mid-gate does not launch a second run. It asks the provider "what became of this key?" and adopts the answer. An unknown outcome is recovered by *observation*, never by *relaunch*.

9.5. Why a Disposable Machine at All

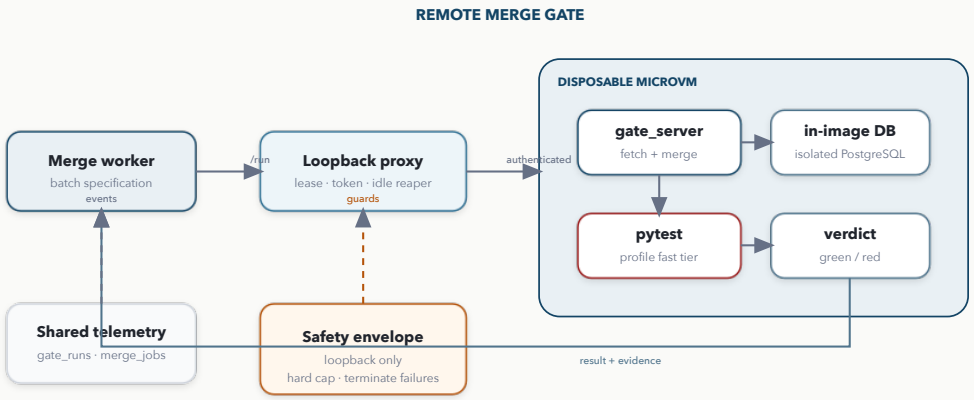


Figure 9.1.: The MicroVM merge-gate path and its safety envelope.

It is worth pausing on *why* the demanding path pays for a whole virtual machine per gate. A local gate shares the operator's CPU, memory, filesystem, and database. On a busy or contaminated host that is slow *and* less trustworthy at the same time — the two failure modes reinforce each other. A MicroVM gets its own kernel, its own checkout, and its own PostgreSQL, built fresh and thrown away. The verdict it returns is a statement about the *code*, because the environment was a controlled constant rather than whatever the host happened to be that afternoon. Disposability is not a cost the design tolerates; it is the point.

9.6. When Disposable Infrastructure Leaks

Disposability has a dangerous asymmetry, and the factory learned it the hard way. **Cleanup can fail silently, while every retry creates another resource.** Over a couple of days in mid-July 2026, a false change in the VM image's identity convinced the pool it needed fresh capacity, while a lock-ownership bug let launches escape the intended single-instance lifecycle. Retries multiplied the leak; a single storm peaked at sixty-four orphaned MicroVMs before it was contained.

The response was not one clever fix but a layered one, because no single control is trustworthy alone: normalize the image identity so a phantom change stops triggering re-warms; track pool warmth independently of the image string; make each launch own its lock correctly and terminate any instance that fails before it is adopted; reap idle shared instances after five minutes; refuse non-loopback URLs; and — the backstops — a **hard fleet launch cap** and a separate **high-water tripwire** that raises an alert when the fleet looks abnormal.

> **Factory Floor — Bound resource creation *before* retrying.** A retry loop wrapped around a leaky create operation is a cost amplifier. The cap must sit *before* creation, and a failed adoption must terminate the resource it just made. Otherwise "try again" is indistinguishable from "leak faster."

9.7. When the Safety Net Reads the Wrong Contract

The cap and the tripwire both depend on one humble function: *count the running MicroVMs*. And that function is where the chapter's original edition found its most dramatic bug — a mismatch where the counter read one response key while the cloud API returned another, so it could count *zero* while instances were live, quietly defeating both backstops.

That bug is real history, and it is instructive, but in current code it is **fixed** — both sides read the same key, and a regression test now pins the contract so the two call sites cannot drift apart again. The durable lesson is not the bug; it is *why it was possible*: a safety mechanism is only as strong as the live API contract it actually reads, and the existence of a constant or a passing mock proves nothing. The fix that mattered was making the contract a *test*.

Two related concerns are honest to keep, because they are still open by design choice rather than defect. The count still **fails open**: if the fleet-list call itself errors, the counter cannot know the fleet size and declines to block — an unknown fleet is treated as a safe one, which is the opposite of what a cap should assume. And the count is **fleet-wide**, not filtered to the gate image, so unrelated workloads in the same account could in principle exhaust the cap. Both are decisions an operator still owes an explicit answer to; the chapter names them rather than hiding them.

(A related caution: remote red-batch isolation — gating several rejected candidates in parallel — is kept disabled by default, because the single-instance proxy's shutdown model isn't safe for it yet. The code path existing is not evidence its resource-ownership model is sound.)

9.8. When the Image Is Stale

One last deviation, subtle and provenance-shaped. The MicroVM runs from a *built image*, and that image is deployed code. Merging a change to the gate server itself does not rebuild an image that already exists. The current rebuild check compares dependency locks and schema versions, but it does not hash the gate server and its recipe — so repository truth can advance while the executing gate environment stays frozen on older code.

The consequence is not a wrong verdict so much as an unanswerable question. A gate can be perfectly isolated, perfectly repeatable, and still leave an operator unable to answer the simplest thing: *which gate code produced this verdict?* The fix — not yet adopted — is a provenance chain: every image carries a source SHA and recipe digest, and every verdict records both. Until then, "the gate is green" carries a quiet asterisk about *which* gate.

A trustworthy green verdict is what finally lets the controller advance Git — the exact candidate becomes `main`, and the note is `done`. And yet, as the next chapter insists, `done` still does not mean that a single user, anywhere, can run the change. Landing code and delivering it are different facts, and the distance between them is the last boundary the factory has to cross.

10 Merge Is Not Deploy

Git can prove that code landed. Only the runtime can prove that a user can exercise it. These are related facts, and neither implies the other.

For the diagnostic command we have followed since Chapter 1, a green merge means its source is now on `origin/main`. It does *not* mean the running `yesod` on any machine has that source. The installed program is executing from wherever it was last deployed; a long-lived service is still running the process it was started as. The note can be `done` in the source lifecycle while the feature remains, to every actual user, invisible.

Closing that gap is deploying, and deploying a live factory to itself is genuinely dangerous — more so than merging, in one specific way: the failures are *partial*. Consider what "just restart the services on the new code" has to survive:

- The change includes a database migration. It gets applied on two hosts, and the third is briefly unreachable. Now the fleet is running one schema against another.
- A service is told to restart on the new release, and it does — but it comes back reading its *old* checkout, and nobody notices because it is up and answering.
- Half the fleet flips to the new release, a health check on the fourth host fails, and now production is split across two versions, both live.
- The deploy fails partway and rolls back — and the rollback deletes work that had already succeeded.

A merge that goes wrong stops a lane. A deploy that goes wrong can leave *production itself* in a state no one can describe. So Yesod does not deploy itself with a hook that fires after a merge. It deploys with a **promotion**: a single durable transaction that moves the entire host fleet from one exact release to another, or refuses and leaves the fleet exactly where it was. This chapter walks one promotion through — the happy path, then the deviations — after the vocabulary.

10.1. The Vocabulary of a Promotion

- **Promotion.** One durable, restart-safe, fleet-wide transaction that moves every host from a `prior_sha` release to a `candidate_sha` release. It is *not* a per-tool action and *not* a hook; it is one row, with a phase, that either reaches `fleet_complete` or rolls back.
- **Immutable release.** The exact content of one git SHA, staged read-only on disk at `/srv/yesod/releases/<sha>` — directories and files made root-owned and unwritable. A release is not a checkout you can drift; it is a frozen artifact.
- **Release markers.** Two files that make a release *provable*: `RELEASE_SHA` (the commit it contains) and a canonical, fixed-format `RELEASE_COMPLETE` marker written *last*,

as the atomic signal that staging finished and verified. A tree without a valid `RELEASE_COMPLETE` is not a release; it is a half-copy.

- **The `current` pointer.** An atomic symlink, `/srv/yesod/current`, that says which release the services actually run. Deploying *is*, at its core, flipping this pointer and restarting — but only after everything that makes the flip safe has been proven.
- **`ops-current`.** A *second*, separate pointer for the conductor's own ops tooling, which must **not** drift during a promotion. Keeping the app pointer and the ops pointer distinct means the machinery running the deploy is not the machinery being deployed.
- **The Conductor and the fleet.** The **Conductor** (host `yesod-dispatch`) runs the control plane and the promotion worker; the **fleet** is the Conductor plus the three coding runners. A promotion acts on all four.
- **Admissions fence.** The same merge-intake pause from Chapter 8 — held for the *entire* promotion, so no new merge lands while the release the fleet is converging on is being moved.
- **Typed effect (intent before effect).** Every external action a promotion takes — stage a host, run a migration, flip a pointer, restart a service — is journaled as a typed row *twice*: an **intent** written before the act, an **observation** written after. A crash between the two is recoverable, and a replay is idempotent, because the journal already knows what was about to happen.
- **Drain.** A request that a runner stop *starting* new work, so it can be safely restarted. A drain never kills live work; it waits for it.
- **`fleet_complete`.** The single terminal fact that means the promotion fully succeeded: every host is proven to be running the candidate. It is set only when every effect and every host has succeeded — never optimistically.

The one to hold onto is **intent before effect**. A deploy is a sequence of irreversible acts against real machines; the only way to make it crash-safe is to write down what you are *about* to do before you do it, and what actually happened after.

10.2. The Good Case

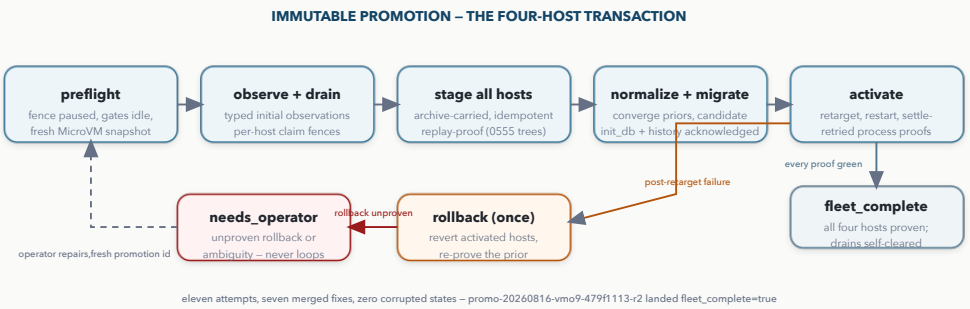


Figure 10.1.: The four-host promotion transaction.

An operator has a reviewed candidate — a SHA that is already on `main` — and wants the fleet

to run it. They submit a promotion. From here it is the worker's transaction, and it runs in order.

First, **observe**. The worker records, per host, what release it is currently on and what its markers say — as typed effects, so the starting state is durable, not remembered. Then it **acquires the admissions fence**: merges are paused for the duration, and the promotion owns the pause token. Then it **drains** the runners, so no runner begins new work mid-flip; live work is allowed to finish, not killed.

Now the careful part, and the ordering is the whole trick: **stage everywhere before activating anywhere.** The worker stages the immutable release — writes `/srv/yesod/releases/<candidate>`, read-only, finalized with a valid `RELEASE_COMPLETE` — on *all four hosts* first. Only once every host holds a complete, verified copy does it touch a single pointer. Staging is the reversible part; activation is not, so all the fragile, network-dependent work happens while nothing is committed yet.

With every host staged, it runs the **migration** once against the authoritative database — but only after a quiescence check proves admissions are paused, no gate is running, and the MicroVM fleet is empty, so nothing is racing the schema change. Then it **activates**, host by host, coding runners first and the Conductor last: for each host, atomically flip `current` to the candidate release, restart the consumers, and **prove** they came back — that the restarted process is actually executing the candidate release, sampled from `/proc`, not merely "up."

When every host is proven, the worker sets **`fleet_complete`**, clears its drains on the way out, and resumes admissions. The fleet is now, provably, running the candidate. Merges start flowing again.

That is a clean promotion: observe, fence, drain, stage-everywhere, migrate, activate-and-prove, complete. Every step left a typed intent-and-observation behind it. Now the deviations — and this is a transaction, so the deviations are where its character lives.

10.3. When a Host Is Still Busy

A runner is mid-task when the promotion reaches it. The worker does not kill the task and it does not skip the host. It has already installed a **drain**, so the runner starts nothing new; the promotion simply **defers** that host — stages it, but declines to activate it while live work is running — and the transaction does not declare success it has not earned. Deferral is a first-class outcome, not an error: a promotion that could not safely finish every host stops short of `fleet_complete` rather than lying about a fleet it only half-moved.

10.4. When a Proof Fails After Activation

This is the case the whole design is built around. The worker flips a host to the candidate, restarts it, and the restart *proof* fails — the process did not come back on the candidate release.

Because activation only ever follows a proven stage, and a failed proof only ever *stops*, the fleet is never left silently split. The worker enters **rollback**, exactly once: it converges the already-flipped hosts back to the `prior_sha` release — which is still staged, still immutable, still proven-good — restarts, and re-proves *that*. The promotion ends `rolled_back`, with `fleet_complete` explicitly false. Production is back where it started, and every step of the retreat is itself journaled and proven. The rollback is bounded to a single attempt; it cannot thrash.

> **Invariant — A failed proof stops; it never guesses forward.** Activation follows a proven stage; a broken proof triggers a bounded, proven rollback to a release known to work. The fleet is only ever on a release the transaction can *prove* it is on — never on a hope.

10.5. When a Proof Fails Before Activation

If a check fails *before* any pointer has moved — a host won't stage, the migration preflight refuses, quiescence can't be established — there is nothing to roll back. The promotion parks at **`needs_operator`**: a terminal state that hands a human a fully-journaled situation to resolve, rather than an agent improvising against production. A promotion that has mutated nothing releases its drains and admissions on the way out; one that has mutated something keeps them, so the fence stays closed until an operator clears it.

10.6. When the Worker Itself Crashes

The promotion worker can die at any point — mid-stage, between the migration intent and its observation, halfway through activating the fleet. Recovery is the same principle as the merge controller's: **re-prove, don't remember.** A restarted worker reads its typed-effect journal, recomputes where it actually got to from the intents and observations already written, and continues or settles from there. Because every effect carries an idempotency key and an intent-before-observation pair, a resumed worker never double-applies a migration or double-flips a pointer. There is exactly one unstarted request at a time, and it is journal-authoritative.

10.7. The Part That Is Still Each Tool's Contract

All of the above is Yesod deploying *itself* — a four-host fleet with a shared database, which is why it needs a transaction this heavy. Most tools are not that. For an ordinary tool, "deploy" is whatever its refinery profile declares: `none` for a library, a restart hook for a small service, a static asset push for a site. Those per-tool hooks still exist and still run for the tool's own repository. So the chapter's original point survives intact — *deployment is part of each tool's contract, not a hardcoded action* — it is only Yesod's own deployment that graduated from a hook into the promotion transaction, because Yesod's blast radius is the factory itself.

10.8. Proving Delivery

Pull the threads together and "someone is actually running the merged code" stops being a hope and becomes an equality the promotion enforces at several layers:

`RELEASE_SHA` marker == `current` symlink target == the git commit.

The worker proves the marker's recorded SHA equals the candidate; proves the staged archive's own commit id equals it; flips `current` and then re-reads the live pointer to confirm it resolves to that release; and samples each restarted process to confirm it is executing from it. Only when all of those agree, on every host, does `fleet_complete` go true. A robust delivery proof therefore chains four things — the candidate note and branch, the merged `origin/main` SHA, the promotion that staged and activated that exact SHA, and the running process observed on it — and until that chain is complete, `done` means *source-complete*, not *delivered*.

This closes the journey from intent to merge to deployment: a change is captured, planned, dispatched, executed, gated, merged onto shared history, and finally *proven* to be running in production — with a refusal, and a durable reason, available at every boundary it could have failed. Part III turns from moving one change through the factory to watching and operating the factory as a whole.

11 Seeing the Whole System

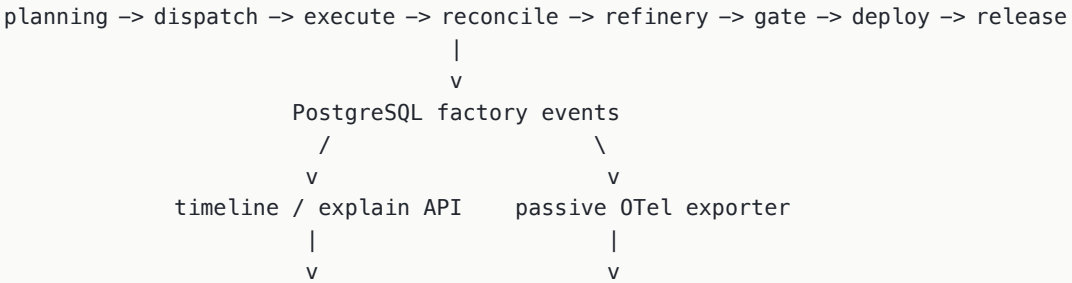
Yesod is observable through several surfaces because no single table contains the whole truth. The next step is not to invent one perfect dashboard. It is to give every important transition a common, causal observation record while preserving the authority of the systems that already own the facts.

> **Deployment Status - Observation Framework.** This chapter began as the normative design recorded in `ys-yes-rklm` and its implementation root `yes-qzogc`. That program has since merged and shipped: the fleet release `479f1113` (promoted 2026-08-16) carries the event envelope, the writer-side emitters, the timeline/explain surface, and the passive exporter. The exporter was commissioned as `yesod-factory-exporter.service` on `yesod-dispatch` on 2026-08-16 and delivered the full event history (13,951 events at commissioning) to the temporary SigNoz backend tracked by `ys-yes-s3gf / yes-xbs08`. One caveat: the refinery/gate emitters (`yes-qzogc.3`) had produced no production events at commissioning time, because no refinery attempt had run since the emitting release activated — the first post-release admission proves or falsifies that path.

11.1. The Observation Decision

The target framework has three layers:

1. **PostgreSQL is the authoritative history.** An append-only, schema-versioned factory event envelope records the causal facts needed to explain a note from planning through release.
2. **OpenTelemetry is the observation language.** A passive controller-host exporter projects committed events and bounded operational state into OTel logs, linked spans, and low-cardinality metrics.
3. **SigNoz and ClickHouse are diagnostic infrastructure.** They may lag, sample, expire, move to a different host, or be replaced without changing factory truth.



```
UI and agents        SigNoz + ClickHouse
```

This is intentionally not "put every log in ClickHouse." The framework joins authoritative lifecycle facts, bounded evidence references, and disposable telemetry without making a dashboard part of the control plane.

> **Invariant - Telemetry is downstream of truth.** An OTel outage may increase exporter lag, but it must never block a factory transaction or change a lifecycle result.

11.2. The Existing Evidence Map

The operator still selects the surface that owns the question:

Question	Primary evidence
What work was requested?	tool note and comments
What is ready next?	Beads workspace and dependencies
Why was this model chosen?	`agent_dispatch_changes.reason` and routing log
Is a runner alive and eligible?	runner registry and heartbeat
What did one attempt do?	run ledger, archived log, progress events
Did it produce a branch?	Git remote ref and result commit
Where is integration?	merge queue, merge job, refinery worker phase
What happened in the gate?	gate attempt, gate log, verdict, evidence
Did the commits land?	`origin/main` ancestry and merge summary
Did deployment run?	merge-job deploy outcome and hook log
Is the service live?	runtime health and reported version

A fleet page may summarize queue depth, but the queue row owns integration status. A note page may show a result commit, but Git owns ancestry. Factory events add a causal index across these authorities; they do not replace them.

11.3. The Factory Event Envelope

Every event has a monotonic ID, schema version, stable namespaced event type, role, source and idempotency key, occurrence and recording time, note and root Bead identity, actor, bounded outcome, reason code, and allowlist-only payload. Its role is one of:

Role	Meaning
`intent`	Yesod committed to attempt an effect owned by another system
`transition`	an authoritative same-database state change committed with the event
`observation`	Yesod observed a fact owned by Git, Beads, a provider, or a deployment

For a PostgreSQL state mutation, authority and event commit or roll back together. For Git, Beads, deployment, and other databases, durable intent and observation events bracket the external effect. The framework never pretends that a network call was atomic with a database transaction.

Optional identifiers are attached only when their authority truly knows them: plan fingerprint, planning job, run, child, worktree, refinery attempt, lane, gate, deployment, host, Git SHA, evidence, causation, trace, or span. Unknown values remain absent. The exact taxonomy and payload allowlists belong to implementation child `yes-qzogc.1`, so prose does not become a competing event registry.

11.4. Coverage Without Invented History

A stage is required only when existing authority proves that the note entered, or should have entered, that stage. An optional stage that was never entered is not a gap. Emitter coverage is registered per schema version, workflow stage, source system, and emitter so staggered deployment remains explainable.

The authoritative explain projection returns one of four states:

State	Meaning
`complete`	every authority-proven required stage is observed
`partial_legacy`	the chain is consistent but predates required emitter coverage
`empty_current`	all required emitters were covered and no qualifying transition exists
`unknown`	evidence or coverage is contradictory, invalid, or unavailable

A nonexistent note remains HTTP 404. `unknown` is not a substitute for "not found," and missing historical telemetry is not silently converted into success.

11.5. Human and Agent Read Models

The authoritative read surface exposes separate bounded endpoints:

```
GET /api/notes/{note_id}/timeline
GET /api/notes/{note_id}/explain
```

Timeline is monotonic and cursor-paginated. Explain reads a snapshot and returns coverage, missing required stages, and bounded machine-readable reasons. The existing note snapshot may advertise compact history status and links, but it does not absorb an unbounded event array.

Agents should use a narrow authenticated API or MCP facade, not scrape the SigNoz UI. Safe operations include searching failed runs, fetching a trace, searching bounded error logs, aggregating a fixed metric over a finite window, and comparing a known service metric against an earlier window. PostgreSQL answers lifecycle questions; the telemetry backend answers diagnostic and aggregate questions.

11.6. OpenTelemetry Projection

The initial spans are bounded around real operations:

```
yesod.plan
yesod.dispatch
yesod.child.run
yesod.refinery.attempt
yesod.gate
yesod.deploy
```

Long asynchronous stages use span links rather than one trace held open for days. OTel logs carry only scrubbed event fields and approved evidence references. Correlation identifiers belong in traces and logs, never as metric labels.

The first metric set is deliberately small: queue age, first stuck-run age, fresh-to-stale heartbeat age, MicroVM active and capacity gauges, deployment skew, and exporter lag. Dimensions come only from finite configured sets such as environment, workflow stage, outcome class, model tier, runner pool, and host role. Note IDs, Bead IDs, run IDs, SHAs, raw branches, model names, hostnames, and free text are forbidden metric labels.

The exporter runs on the controller service host, not in workers or gate guests. It reads committed events in order and advances a per-sink PostgreSQL checkpoint only after acknowledgment. Delivery is at least once: a duplicated diagnostic span is safer than an advanced checkpoint with lost data.

11.7. Private Content and Network Authority

The durable payload is allowlist-only and capped. Factory events and OTLP must never contain prompts, model responses, tool bodies, credentials, environment dumps, or raw command output. Those remain in the protected Blob/NAS evidence path and are linked by bounded references such as content hash, media type, byte count, and artifact reference.

Network authority is narrow:

- workers and MicroVM guests receive no Vultr SSH or telemetry credentials;
- only the controller exporter needs Tailscale access to OTLP/HTTP 4318;
- operators may receive separate UI access to port 8080; and
- ClickHouse, Keeper, and PostgreSQL are not published as host ports.

Monitoring must also remain behaviorally passive. Status polling must not launch a MicroVM, reset its idle timer, retry work, or mutate a note.

11.8. Current Windows Into the Factory

The current CLI still provides useful read-only summaries while the unified history is being built:

```
yesod factory status
yesod fleet status --pretty
yesod codefactory-dispatch status
yesod codefactory-dispatch queue
yesod runs list
yesod runs show RUN_ID
yesod refinery status --pretty
yesod refinery list
yesod main-ci status
yesod agent roster
```

The run ledger opens at claim and finalizes once. Separate progress heartbeats make buffered worker activity visible, while the finalized row remains the attempt ledger. Integration similarly spans mutable queue state, immutable attempt and evidence records, worker heartbeats, gate results, and durable merge summaries. The framework gives these records a common causal index without flattening their retention or ownership differences.

> **Terminology Trap - Queue status is not note status.** A note can be `awaiting_merge` while its queue row is `gate_red` or `blocked`. Debugging only one side produces false conclusions.

11.9. Temporary SigNoz Deployment

[deploy, 2026-08-13] `obs-vultr` in Vultr's Seattle region hosts the temporary diagnostic backend while the planned PowerEdge host is unavailable. SigNoz, its OTel collector, Click-

House/Keeper, and PostgreSQL are pinned; UI 8080 and OTLP 4317/4318 bind only to the node's Tailscale address.

Workspace activation enabled the real ClickHouse pipelines. A synthetic OTLP trace for service `yesod-observability-smoke`, span `vultr.signoz.smoke`, and attribute `yesod.note.id=ys-yes-s3gf` returned HTTP 200 and was verified in the trace index. This proved the temporary sink; the production proof followed on 2026-08-16, when the commissioned exporter delivered the complete factory event history and ClickHouse counts matched the PostgreSQL checkpoint exactly.

Moving the sink to the PowerEdge requires only an endpoint change — and the checkpoint design makes the migration better than a cutover: give the new backend a fresh `sink_id` and the exporter replays every event ever recorded from PostgreSQL, so the permanent backend starts life with full history. The Vultr node is genuinely disposable.

The backend now carries three standing artifacts beyond the raw signals: two dashboards and an alert. Every panel embeds its own operating knowledge — a hover tooltip stating what the panel measures, what good looks like, what bad looks like, and the first response — so the dashboards teach their own interpretation instead of assuming an operator who already knows.

The `Yesod Factory` dashboard is the overview: pipeline health on top, factory activity below. Reading the capture above, top-left to bottom-right:

- *Exporter Lag* is flat at zero except one spike — the backlog that accumulated while the exporter's first, session-hosted incarnation lay dead, draining in under thirty seconds at commissioning. A healthy pipeline is a boring flat line; the spike is the incident that justified the alert.
- *MicroVMs: Active vs Capacity* shows gate verification bursts against the capacity ceiling — and the ceiling itself **stepping from 8 to 16** at the moment the stale `YESOD_FACTORY_MICROVM_CAPACITY` value was corrected. Before the fix, active spikes visibly pierced the claimed ceiling: a configuration-fed gauge asserting something false about the world.
- *Signal Events by Type* is the factory heartbeat with the reconciler noise excluded — planning, dispatch, closes, and deploys as distinct colored series; the bursts line up with the promotion campaign and the day's planned-root runs.
- *root_close Noise Rate* charts bug `ys-yes-x0yk` on purpose, at hundreds of events per minute. When the fix merges, this panel will draw a cliff to zero — the dashboard is pre-instrumented to witness its own bug fix.
- The lower rows — note transitions, child-close intent-versus-observed moving in lockstep, runs started-versus-completed, workflow-stage mix, outcome classes, deploy observations — are the lifecycle contracts as time series.

The `Yesod Contract Integrity` dashboard asks one question eight ways: *are the contracts holding?* Each panel pairs a claim with its evidence — the full close triplet (intended, observed, completed) overlapping exactly; agent-claims versus ledger-confirmed completions (the no-op-agent detector: a gap here is precisely the failure shape the contracts were built to catch); dispatch arms versus route changes (the lone route-change spike is the provider-outage rewiring, recorded as it happened); worktrees versus runs; non-success outcomes; and a deliberately empty panel — *Refinery & Gate Events (awaiting first witness)* — whose

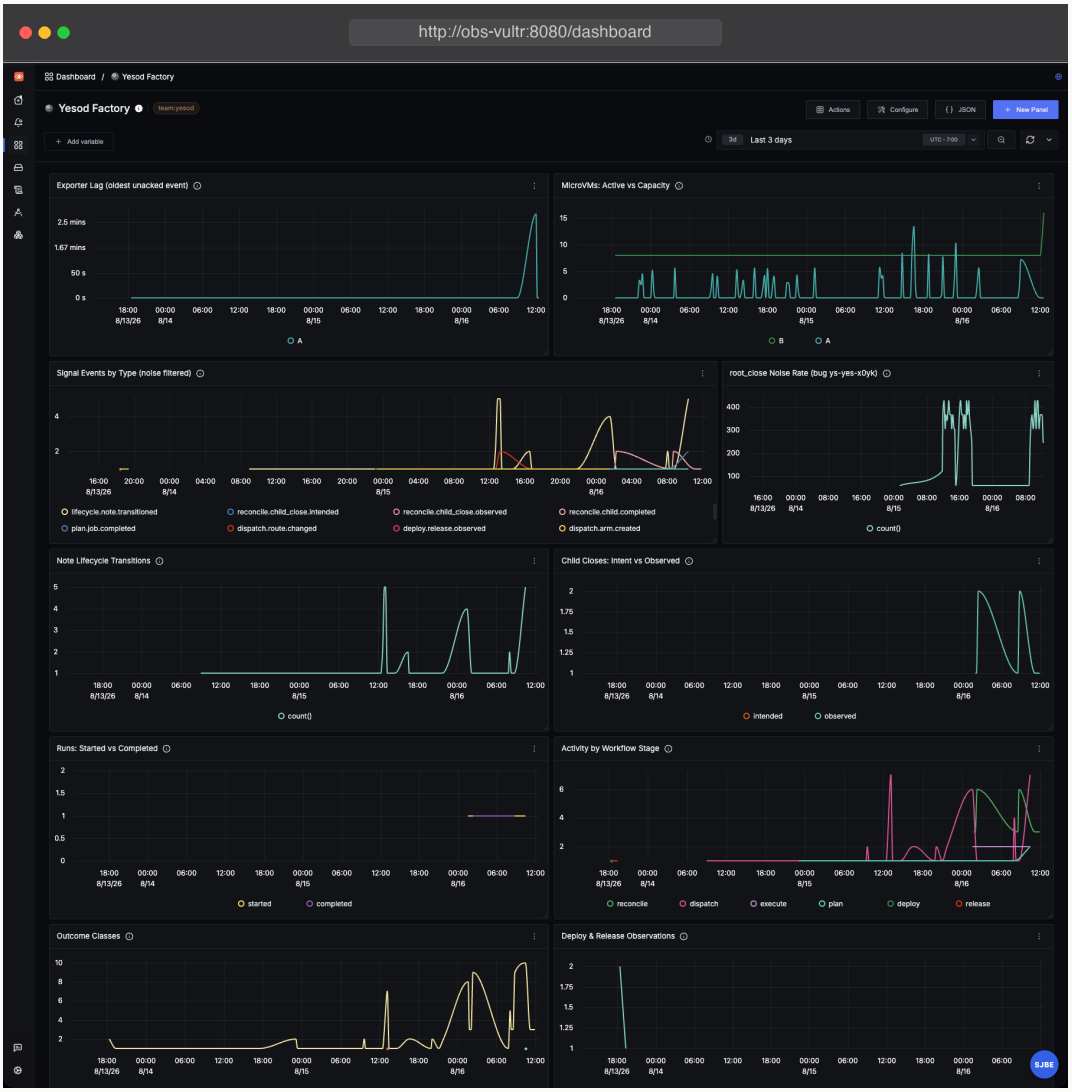


Figure 11.1.: The Yesod Factory dashboard over its first three days of production.

Figure 11.2.: The Contract Integrity dashboard: every panel pairs an intent with its confirmation.

tooltip explains that those emitters are merged but unproven until the first post-release admission draws a line on it. An empty chart, honestly labeled, is a coverage statement; the same chart silently absent would be a lie of omission.

The exporter-lag alert (`Yesod factory exporter lag high`) fires when `yesod.factory.exporter.lag` averages above 300 seconds over five minutes. It encodes the first real incident: the exporter initially ran only as a foreground process in an agent session, died silently when that session ended, and the lag grew unnoticed until commissioning. A dead exporter never blocks the factory — but it must never again go unnoticed.

11.10. Delivery Program and Evidence

The accepted plan had five delivery boundaries, all now merged and closed (root `yes-qzogc`, closed 2026-08-14). The graph split its first boundary into four smaller implementation tasks, for ten issues including the root:

Order	Bead	Boundary
1	`yes-qzogc.1`	parent boundary for event and coverage contract
1a	`yes-qzogc.1.1`	additive PostgreSQL event schema
1b	`yes-qzogc.1.2`	versioned contract and append-only writer
1c	`yes-qzogc.1.3`	targeted event-contract tests
1d	`yes-qzogc.1.4`	replay and coverage documentation
2a	`yes-qzogc.2`	planning, dispatch, runner, run-ledger, reconciliation emitters
2b	`yes-qzogc.3`	refinery, gate, deployment, release emitters
3	`yes-qzogc.4`	timeline/explain API and Single Note UI
4	`yes-qzogc.5`	passive OTLP exporter, metrics, credentials, runbook

Children 2a and 2b may proceed independently after the event contract. The read surface requires both emission branches; the exporter follows the authoritative read surface.

Figure 11.3.: Panel tooltips carry the operating knowledge: what it measures, what good and bad look like, and the first response.

Yesod reference	What it establishes
`ys-yes-rklm` / `yes-qzogc`	normative Observation Framework and implementation program
`yes-qzogc.1` through `.5`, plus `.1.1` through `.1.4`	delivery graph and dependency order (all closed)
`ys-yes-s3gf` / `yes-xbs08`	temporary Vultr SigNoz backend, security boundary, smoke proof
`ys-fac-gb9i`	durable planner handoff for the architecture split
`ys-yes-x0yk`	emitter noise bug found on day one of live export
`ys-yes-d2le`	follow-up: promotion-transaction spans and refusal-code events

The full normative contract, rollout rules, exact initial metrics, acceptance criteria, and evidence commands are in `reference/observation-framework.md`. The operational companion — where to look, how to query, how to commission, verify, and migrate the exporter — is `reference/observability-field-guide.md`.

11.11. What the First Live Days Taught

Observability observed itself immediately, and the first lessons arrived before the first dashboard did:

- **The ledger caught the sink's death.** The exporter's first incarnation was a foreground process in an agent session; when the session ended, delivery stopped, and nothing noticed for hours. The checkpoint row preserved the exact acknowledgment boundary, so commissioning the systemd unit resumed delivery without loss — a 4,226-event backlog drained in under thirty seconds. The lag alert exists so the *silence*, not just the failure, is visible next time.
- **The first emitter bug was a contract-spirit violation, not a crash.** 99.2% of all events were `reconcile.root_close.observed`, emitted once per reconciler poll over already-closed roots — an observation of nothing changing, hundreds of times per hour. Filed as `ys-yes-x0yk`. The lesson generalizes: *observation events must record observations of change, or first observations — never the heartbeat of a poll loop.* The dashboard charts the noise rate on purpose; the fix will be visible as a cliff.
- **Workers emit, but only the ledger talks to the sink.** Events arrive with actor identities from every runner, the reconciler, operators, and agents — yet every one of them is a PostgreSQL row first, and only the single controller exporter speaks OTLP. Worker hosts hold no telemetry credentials. The topology held exactly as designed.
- **Static config drifts; measured config doesn't.** The MicroVM capacity gauge reported 8 while the active-guest gauge peaked at 16 — the exporter's environment file predated

the gate-fleet expansion. Trivial to fix, but a reminder that a gauge fed by configuration is a claim, not an observation, and claims need the same review discipline as code.

- **Coverage claims need production witnesses.** The refinery/gate emitters merged with the rest of the program, but zero `refinery.*` or `gate.*` events existed after commissioning — not a bug, just no refinery activity since the emitting release activated. Until the first post-release admission produces them, that path is *merged*, not *proven*. The distinction is the whole reliability-contracts lesson in miniature.

11.12. A Read-Only Diagnostic Rhythm

Until the timeline can assemble this chain directly, diagnose a stuck item in pipeline order:

1. identify the note and root Bead;
2. inspect note status, disposition, hold, retry time, and dependencies;
3. inspect Beads readiness and blocker integration;
4. inspect runner eligibility and live attempts;
5. inspect the remote branch and result commit;
6. inspect the queue row and refinery attempt;
7. inspect gate progress, verdict, and evidence;
8. inspect `origin/main` ancestry; and
9. inspect deploy outcome and runtime version.

> **Operator Check - Preserve the scene.** Before a reset, kill, dequeue, or retry, capture the run row, queue row, branch state, gate tail, and relevant service status. Recovery that destroys causality makes the next incident harder.

12 Failure Is a State Machine

In a factory run by people, failure is an interruption: something breaks, someone notices, someone decides what to do. In a factory run by agents, nobody notices unless *noticing is a mechanism* — and nobody decides safely unless the decision is a state the system can be in, with defined ways out.

So Yesod does not treat failure as an exception to the happy path. It treats failure as **a set of states a piece of work can occupy**, each with a name, an entry condition, and a defined exit. "Stuck" is not a mood; it is a row. This chapter is the map of those states — where work can fail, what each failure *is*, and what moves it forward — organized by the three boundaries a change crosses: execution, integration, and deployment.

The deep reason this matters is a single asymmetry, which is worth stating before the details:

> **The components fail constantly; the guarantees must not fail once.** Agents no-op, networks drop, providers exhaust their credit, processes die between two lines. The point of a failure *state machine* is that none of that turns into a corrupted merge or a half-deployed fleet — the failure is *absorbed* into a named state, not leaked into shared truth.

12.1. The Vocabulary of Failure

- **Verdict vs. classification.** Two independent questions about any failed execution: *what happened to the work* (a verdict: complete-but-uncommitted, partial, no-progress, infrastructure) and *why the attempt ended* (a classification: timeout, killed, runaway, no-op, and so on). A retry policy that reads only an exit code confuses these; Yesod records both axes.
- **`awaiting_verification`.** The fail-closed parking state. When an execution ends without provable success — no result commit, an ambiguous outcome — the note parks *here*, carrying a structured reason, rather than being called done or blindly retried. It is the system refusing to guess.
- **`blocked`.** Work stopped by a cap — too many retries, or repeated no-progress — that needs *changed conditions* or human judgment before it is worth trying again.
- **Immutable attempt.** On the merge side (Chapter 8), a failure is a *terminal attempt* — `gate_red`, `push_raced`, `needs_rework`, `infrastructure_failed`. Terminal attempts never reopen; a retry is a new attempt. So "failure" there is literally a state you cannot transition out of, only supersede.
- **Operator inbox.** The deterministic recovery surface. When automated recovery would have to *guess*, the controller instead opens a durable item for a human — a queue of decisions, not an agent improvising against production.

- **Fail closed.** The governing principle beneath all of it: when evidence is missing, ambiguous, or contradictory, refuse and park with the contradiction named — never continue on an assumption.

12.2. Execution Failures – Where the Work Is Made

The first boundary is the agent run itself, and here the state machine is at its richest, because an agent can fail in the most ways.

The happy path is Chapter 7's: the worker commits, pushes, proves a remote branch, and the note reaches `awaiting_merge`. Every deviation is a defined state instead.

- **The run produces no commit.** This is not one failure but four, which is why the *verdict* axis exists. The model may have done nothing (`no-progress`); it may have written good code and failed to commit it (`work-present-uncommitted`); a required tool may have died before launch (`infrastructure`); or the requested behavior may already have existed. The runner classifies which, because the right next move differs for each — and in the uncommitted case, a blind retry would *destroy the evidence*. The note parks in `awaiting_verification` with the reason recorded.
- **The run makes no progress, twice.** Two consecutive `no-progress` verdicts halt early and move the note to `blocked`, rather than paying for a third evidence-free attempt. Repetition without change is not resilience.
- **The push doesn't land.** A commit that cannot be proven on the remote does not enter the merge path; the note stays parked until a real branch exists, because the next stage runs in another checkout and needs something it can fetch.
- **The infrastructure failed, not the model.** A rate limit or a dead dependency is released *without* consuming the note's retry budget — "the lane couldn't run" is a different fact from "the model couldn't do it," and only the second should burn a retry.

Bounded retries tie these together: each real failed attempt increments a count, backoff grows, a model-capability failure can escalate to a stronger lane, and at the cap the note goes `blocked`. The rule underneath is one line — *retry only when the next attempt is meaningfully different* — and every mechanism above exists to make that judgment without a human in the loop.

12.3. Integration Failures – Where the Work Lands

The second boundary is the merge, and Chapter 8 already walked its failure states in detail; here they matter as *states in the same machine*.

A merge failure is an **immutable terminal attempt**, and that is the key difference from a naive queue. A red gate ends an attempt at `gate_red`; a lost lane ends it at `push_raced`; a conflict ends it at `merge_conflict` and reroutes to a bounded AI repair whose output must re-earn the gate. None of these *reopen* — a retry is always a new, freshly-fenced attempt linked to the old one. So on the integration boundary, "failure" is not a status that might get overwritten; it is a sealed record, and recovery is always forward, into a new attempt.

Two properties keep one failure from becoming many:

- **Lanes contain the blast radius.** A blocked change on `yesod/main` does not stall `ttrac/main` — they are different lanes and were never behind each other. Head-of-line blocking is prevented structurally, not by reordering a queue around blocked entries.
- **Durable scratch is reconciled, not trusted.** The merge queue survives crashes, which means it also accumulates stale rows; the controller's per-tick reconciliation compares queue state against notes, branches, and history and repairs the drift — because persistence without reconciliation turns a transient failure into a permanent obstruction.

When integration recovery would have to *guess* — an ambiguous state no rule can resolve — the controller does not guess. It opens an **operator-inbox** item. This is the single most important correction to the old model: recovery is deterministic SQL plus, at its edge, a human decision — never a standing agent asked to reason its way out.

12.4. Deployment Failures – Where the Work Runs

The third boundary is promotion, and Chapter 10 walked its states: a busy host is **deferred**, a post-activation proof failure triggers a bounded **rollback** to a proven release, a pre-activation failure parks at **`needs_operator`**, and a crashed worker **recomputes** its position from its typed-effect journal. The through-line is identical to the other two boundaries: *a failed proof stops; it never guesses forward*, and the fleet is only ever on a release the transaction can prove it is on.

What deployment adds to the failure vocabulary is the *partial* failure — a fleet split across releases. The design's answer is ordering (stage everywhere before activating anywhere) and proof (activate host-by-host, prove each), so that the split state, if it happens, is transient and self-correcting rather than durable and silent.

12.5. A Note on Host Placement

One failure class is not about state at all but about *where code runs*, and it earned its place the hard way. A watchdog that stops a stuck process by reading its PID from a shared database and calling `os.kill()` is only correct if the watchdog and the process are on the same host. Across hosts, that PID may name nothing — or, worse, an unrelated process. The retired supervisor-era watchdog had exactly this latent flaw, and the redesign's consolidation of the control plane onto one Conductor is partly a response to it.

> **Factory Floor — Host placement is part of correctness.** A watchdog is not host-agnostic merely because it reads its target's PID from a shared table. Either co-locate it, route the signal explicitly, or make a host mismatch a hard refusal. "It has the PID" is not "it can safely kill it."

12.6. The Shape of the Whole Machine

Step back and the three boundaries are the same machine at three scales. At each one, success is a narrow proven path; every other outcome is a *named state* — parked, blocked, terminal-and-superseded, deferred, rolled-back, or handed to an operator — reached by refusing to advance on missing evidence. The states differ; the discipline does not: **name the failure, make it inspectable, and never let it leak into shared truth.**

That discipline is not decoration on top of the factory. It *is* the factory's reliability, and the next chapters — on cost, on the patterns worth reusing, and finally on the reliability contracts themselves — are all, in the end, about what it costs to hold that line and why it is worth it.

13 Cost-Aware Dispatch

Yesod treats model cost as an engineering property, not a billing-page afterthought.

The important unit is not the price of one token. It is the expected cost of one accepted, merged, useful change.

13.1. Capturing Usage Honestly

Native Claude runs can report input, cache, output, and total cost in structured result data. Opencode-backed runs require a different path: each dispatched session is pinned to its own directory and start time, and Yesod reads token counts from the opencode session database.

Input-side accounting includes ordinary input, cache reads, and cache writes. Output-side accounting includes output and reasoning tokens. That convention measures billed work rather than only the most visible prompt and response fields.

Pricing resolution follows a clear precedence:

1. editable rows in the PostgreSQL `model_pricing` table;
2. a source-code fallback table with an `as of` date;
3. unknown.

If a model is missing or any required rate is unknown, tokens can still be recorded but `cost_usd` remains `NULL` with a reason. Yesod does not substitute a guessed price.

> **Invariant — Unknown is not zero.** A missing cost must render as unknown, not free. Zero would reward the least observable lane.

The run ledger stores usage after finalization. Rollups can sum every attempt for a note or a root bead and its children. Failed, timed-out, and killed runs still contribute because they still consumed resources.

13.2. The Wrong Metrics

Several attractive metrics can route work badly.

Price per million tokens ignores how many tokens the lane uses and whether it finishes.

Cost per run rewards short failures.

Commit rate rewards low-value commits and ignores gate quality.

Merge rate is better but can still ignore churn, regression cost, and task difficulty.

Lines per dollar can reward verbosity or generated code rather than maintainability.

No single number should decide every route. But one metric is a much better starting point:

$$\text{cost per merge} = \frac{\text{cost of all attempts in a lane/project bucket}}{\text{number of landed changes attributed to that bucket}}$$

The `stats_data` layer computes economics by project and lane: dispatched runs, merges, merge rate, total cost, cost per merge, churn, net lines, and code per dollar. The leaderboard ranks only buckets with known delivered economics; unpriced or merge-less buckets remain visible but unrankable.

13.3. Cost Awareness Today

Current dispatch is **cost-aware**, but not a fully autonomous optimizer.

The dispatcher primer maps task complexity to lane families and requires each routing reason to explain both capability and cost. The selected target is stored structurally in the eval queue and note; the reason is stored in the dispatch audit trail. Operators can review past decisions with the routing log and correlate them with run and merge outcomes.

A typical rationale has the form:

```
medium bug fix in a familiar repo — lower-cost lane has sufficient
context and a strong merge history here; escalate only on ability failure
```

This is a human/agent policy informed by telemetry. The runner does not currently solve a global optimization problem and silently reassign every note to the cheapest predicted lane. That restraint is healthy while the data remains sparse and task difficulty is confounded with model choice.

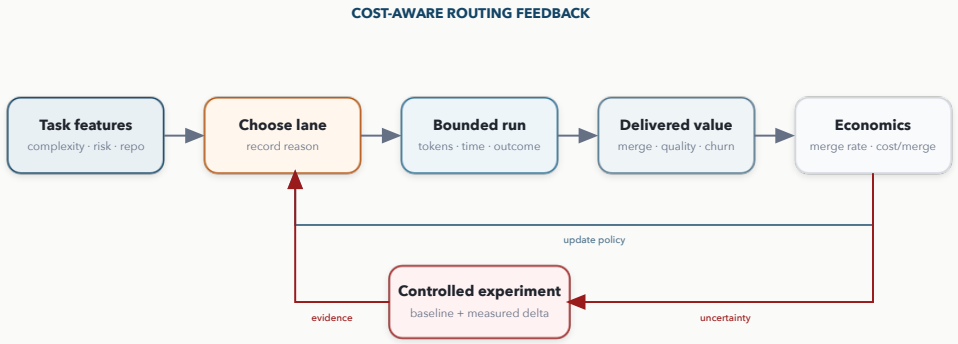


Figure 13.1.: The cost-aware routing feedback loop.

13.4. Expected Delivered Cost

A practical routing estimate should include:

- expected token cost of the first attempt;
- probability of producing commits;
- probability of passing acceptance;
- probability of a green merge gate;
- expected retry and escalation cost;
- gate compute cost;
- expected human triage cost;
- risk-weighted cost of a regression;
- value of latency for the task.

A simple approximation is:

$$E(C_{delivered}) \approx \frac{C_{attempt} + C_{gate} + E(C_{recovery})}{P(merge)}$$

The equation is not a production policy by itself. It makes the hidden assumptions visible. A lane that costs one quarter as much per attempt but merges one tenth as often is not cheaper.

13.5. Bounded Spend Is Part of Routing

Lane choice is only one cost control. Yesod also bounds:

- runtime;
- input-token consumption;
- retry count;
- no-progress repetition;
- gate attempts;
- MicroVM fleet size;
- idle MicroVM lifetime.

These controls handle tail risk. The most expensive incident is often not the normal price of an advanced model; it is an unbounded process or leaking retry loop.

13.6. Project-Specific Policy

Model performance is not globally uniform. A lane may excel in a Python CLI and fail repeatedly in a large frontend or concurrency-heavy service. The correct comparison unit is therefore often **lane × project × task class**.

Useful task features include:

- repository and language;
- complexity and file breadth;
- change type;
- prior context length;
- database/concurrency involvement;
- acceptance strength;
- historical lane success;
- recent rate limits or outages.

This is why Yesod records the routing reason. Without the features that motivated the decision, the outcome cannot teach the next decision.

13.7. Exploration Without Gambling the Factory

An optimizer needs exploration or it will never learn that a cheaper lane improved. The mad-scientist role provides a safer pattern: controlled experiments with a baseline, a measured delta, and a sandbox.

For example:

- compare two lanes on matched medium-sized visualization tasks;
- measure commit rate, gate-green rate, wall time, and cost per merge;
- exclude tasks whose difficulty is not comparable;
- cap the experiment's spend;
- promote the result into routing policy only after evidence.

This is closer to a contextual bandit than a static price table, but production routing should begin with conservative guardrails and explicit confidence.

13.8. The Next Cost-Aware Step

A mature closed loop would recommend, not merely display:

1. estimate task features and uncertainty;
2. filter lanes by capability, policy, and availability;
3. predict delivered cost and failure risk per lane;
4. choose the lowest-risk option inside a task budget;
5. reserve a small exploration budget;
6. record the decision and confidence;
7. update the model only after merge, deploy, and short-term quality signals.

The word **after** matters. Training the router on "agent exited successfully" would optimize for narration. The value signal belongs at integration and beyond.

Part III.

Building Beyond the Current Factory

14 Patterns Worth Reusing

Yesod is one implementation. Its most useful ideas can be adopted without copying its roles, database schema, or host names.

14.1. Durable Intent Before Automation

Start with an addressable work record. It should preserve the request, acceptance evidence, dependencies, routing decision, attempts, and final disposition.

Do not begin with a swarm. A system that cannot explain why a task exists will become less understandable when ten agents work on it concurrently.

14.2. Separate Work Graph from Execution History

Planning structure and execution history change at different rates.

A work graph answers what depends on what. An execution ledger answers who tried, when, with which model, and what happened. Combining them into one mutable status field makes retries, partial progress, and parallel work hard to represent.

14.3. Make Dispatch Orthogonal

"Ready," "open," and "assigned to a lane" are different facts. Store them separately.

This enables explicit holds, re-routing without rewriting lifecycle, and analysis of why a note was open but invisible. It also lets a plan exist before anyone decides what model should attempt it.

14.4. Claim Atomically, Repair Cross-Store Effects

Use a short transaction and row lock for the authoritative claim. Perform slower external assignment after the lock is released. If that side effect fails, compensate visibly.

Pretending that two independent stores share a transaction creates false confidence. Explicit repair is safer than imaginary atomicity.

14.5. Give Every Run a Disposable Workspace

Worktrees provide isolation without a full VM per coding attempt. The important properties are:

- a known base revision;
- a unique branch;
- no shared uncommitted state;
- a durable path for forensic recovery;
- clear cleanup rules.

The workspace should be disposable, but the commits and logs should not be.

14.6. Put Governance Outside the Model

Runtime, token budget, kill requests, process-tree termination, and remote-branch proof belong in a governing service. The worker prompt should explain the rules, but code should enforce them.

This is the general pattern behind safe agent autonomy: **semantic freedom inside mechanical bounds**.

14.7. Define Proof of Work

Choose proof appropriate to the task:

- code task — commits past a known base;
- data task — a passing acceptance probe;
- TUI task — an external smoke verdict;
- remote contribution — a fork branch and PR URL;
- deployment — a runtime version or artifact digest.

Do not treat exit code or transcript language as universal proof.

14.8. Use One Automated Integration Door

Workers should propose branches. A separate integration service should reset to current main, merge candidates, run the authoritative gate, verify the remote push, and record what landed.

This reduces the number of credentials and contexts that can mutate shared history. It also makes throughput and failure visible at one narrow waist.

14.9. Distinguish Red from Unavailable

A test failure is information about the candidate. A gate transport failure is information about the infrastructure.

They need different counters, different retry policies, and different operator responses. Treating unavailable as red punishes good code. Treating red as unavailable creates infinite retries.

14.10. Reconcile Durable Scratch

Any queue that survives process death needs a repair loop for:

- duplicates;
- missing sources;
- already-completed work;
- reverted work;
- stuck intermediate states;
- blocked-head starvation.

Persistence solves loss and creates rot. Reconciliation is the price of durability.

14.11. Record Reasons, Not Only Decisions

A lane assignment without rationale cannot train a better policy. A manual hold without a reason becomes archaeological debris. A retry without a recorded change invites repetition.

Store the “why” next to the “what” while the decision is fresh.

14.12. Price Delivered Outcomes

Capture tokens and cost per attempt, preserve unknown values as unknown, and roll all attempts into delivered outcomes. Route on expected cost per accepted change, not list price.

14.13. Make Authority a Data-Flow Property

Do not infer authority from a display name or message sender. Define the channel through which human authority can enter, and refuse to upgrade peer coordination into commands.

This principle applies beyond agents. Webhooks, queue messages, and service identities also need explicit trust boundaries.

14.14. Date the Fleet

Source-grounded documentation and deployment documentation are different products. Put a commit on the former and an observation time on the latter.

A current architecture diagram can coexist with a stale runner. The documentation should make that difference obvious.

14.15. 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. **Economize** — measure cost per delivered outcome and route with reasons.
8. **Recover** — add reconciliation, janitors, and host-aware watchdogs.
9. **Adapt** — run controlled experiments and update policy from landed results.

Each stage remains useful without the next. That is important: the system should deliver value before it becomes autonomous.

15 The Reliability Contracts

There is a moment, early in every autonomous factory's life, when an agent says "done" and it is not. Not maliciously — a model exits cleanly after producing nothing, a network blip eats a push, a process dies between the claim and the launch. In a factory run by people, someone notices. In a factory run by agents, nobody notices unless *noticing is a mechanism*.

The last ten chapters were that mechanism, boundary by boundary: a runner that claims exactly once, a worker judged on artifacts not confidence, a merge controller that never guesses, a gate that fails closed, a promotion that proves the fleet is running what it merged. This chapter steps back from the machinery and names the *pattern* the machinery keeps repeating — because it is one pattern, applied five ways, and once you can see it you can see the whole factory as a single idea.

That idea is the **reliability contract**. And the way to understand it is not a definition but a story: a single afternoon rule that grew, failure by documented failure, into a system that promoted a live four-host fleet through eleven attempts without once leaving it in a state it could not prove.

15.1. The Simple Example That Started It

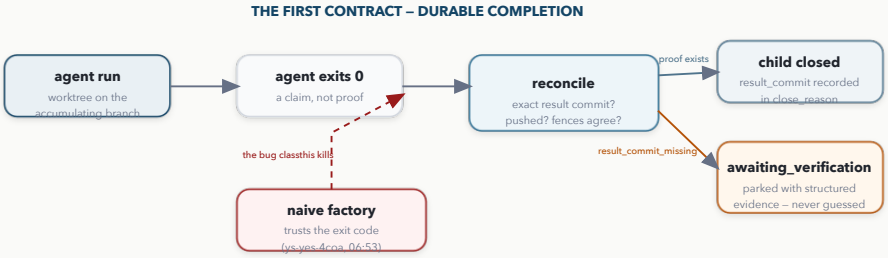


Figure 15.1.: The durable-completion contract.

Consider the smallest possible version of the problem. One planned task, one agent, one branch. The runner launches the agent in a worktree; it works; it exits with status zero. The naive factory does the obvious thing: exit zero means success, close the task.

One morning a coding agent ran for ten minutes on the second child of a planned note, logged nothing, produced zero commits, and exited cleanly. The naive factory would have closed

that child, launched the third on top of nothing, and merged a feature that half-existed — and nothing downstream would ever have looked back.

The first reliability contract is the refusal to let that happen:

> **A child is complete only when an exact result commit exists on the accumulating branch, is pushed, and every identity fence agrees. 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` carrying a structured reason — the missing-commit code, the run id, the child id, the plan fingerprint — and nothing else moves. No third child. No merge. No guessing. This is the pattern every later contract repeats: take an assumption ("agents that exit zero succeeded"), convert it into an *evidence requirement* (a result commit), and make the missing-evidence path a first-class, parked, inspectable state rather than a silent continuation. The factory's word for the whole discipline became **fail closed**.

15.2. The Five Rules, Named

Strip every contract in the book to its moves and the same five appear. They are the real vocabulary of this chapter, so here they are plainly:

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. Not a signal that suggests success: the thing itself.
2. **Fail closed with structured evidence.** When the evidence is missing, ambiguous, or contradictory, refuse — and the refusal names the contradiction. A refusal that says *what disagreed with what* is a datum; a stack trace is an apology.
3. **Journal intent before effect.** Anything that changes the world writes a durable *intent* row before, and an *observed* row after, 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 — never from a cached phase. Restart-safety comes from recomputation, not checkpoints.
5. **Bind to exact identity.** Work is bound to exact SHAs, plan fingerprints, note revisions, idempotency keys, fencing generations. "Close enough" must never match.

You have already watched all five at work. The merge controller's fencing tokens are rule 5; its lease-based recovery is rule 4; the gate's verdict-versus-infrastructure line is rule 2; the promotion's typed effects are rule 3; every "no commit, no completion" park is rule 1. The contracts are not a new subsystem. They are the shape the whole factory already has, made explicit.

15.3. Recovery Is a Contract Too

Parking a failed note is easy. Resuming it safely is where reliability is actually earned, because recovery is where a system is most tempted to guess.

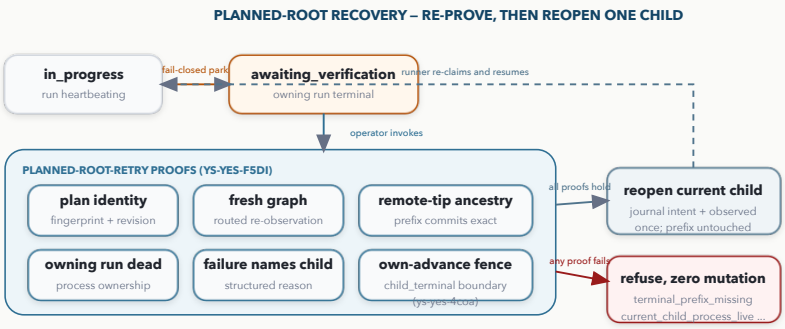


Figure 15.2.: Planned-root recovery.

Take the parked child from before. It had real work behind it — 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 because it is one: re-prove the plan identity, re-observe the routed child graph fresh, verify the remote branch descends exactly from the recorded prefix, prove the owning run's process is actually 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 step aside for a runner to re-claim.

Every proof has a refusal on the other side, and the refusals are the interesting part. 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, a fence added the day a reviewer-reopened child met its own prior push. A claim history contaminated by earlier runs is matched *run-scoped*, never across the child's whole history — the precise bug that once stranded a production note until the matcher was narrowed. And the campaign found recovery's honest edges and *closed* them: a first-child failure has no prefix to preserve, so it now gets its own recovery path rather than a refusal; a failure before any child is selected names no child at all, and is handled as its own case. Each edge got a reproduction, a filed note, and a fix — because a recovery gap you have written down is operational debt, while one you have not is a future outage.

15.4. The Campaign: Eleven Attempts, Zero Corruptions

The contracts above govern source. The final one governs the running fleet, and it is where the story becomes worth a chapter — because the immutable-promotion machinery of Chapter 10, fully specified and built, had *never once run end to end*. Its first real execution became a sequence of eleven attempts, and every single stop was a contract being discovered, filed, fixed through the full factory pipeline, and merged — usually the same hour.

The eleven read as an archaeology of implicit trust. A consumer-health proof compared systemd's exit-code field to 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

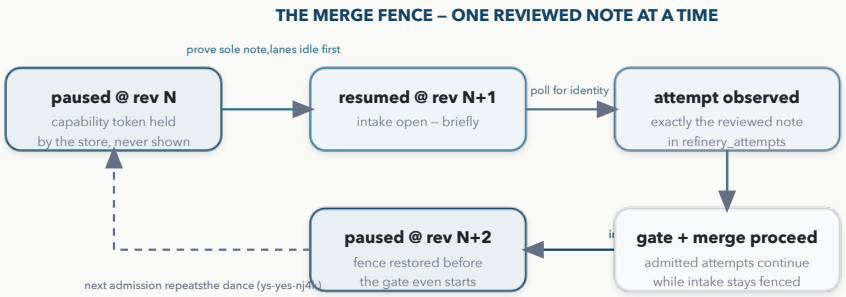


Figure 15.3.: The merge fence and the promotion transaction.

allowlist was missing the worker's first call, and the retarget was hardcoded to one historical release in three separate layers, so the machinery could only ever re-promote its own past. The worker overwrote its own initial host observations with drain bookkeeping and then could not re-read what it had destroyed — fixed by making observations typed, append-only effects. The live schema held six migrations applied before the migration framework existed, and the preflight correctly 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 — fixed with a bounded settle-and-retry, and with a rule that every proof emit exactly one structured result even on failure.

Twice — attempts eight and ten — the transaction 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 ever follows a proven stage and a failed proof only ever stops. Attempt eleven ran the whole transaction — observe, drain, stage everywhere, migrate with acknowledged history, activate with settle-retried proofs on all four hosts — and terminated `fleet_complete`, clearing its own drains on the way out. The fleet that had drifted well behind its own `main` was now running a release cut that day, deployed by machinery that had just finished proving *itself*.

The deepest lesson of the campaign is in that asymmetry. 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 — and the ledger shows zero bad merges, zero corrupted lifecycle rows, zero half-promoted fleets. *The components failed constantly. The guarantees did not fail once.* That asymmetry is the entire point, and it is only achievable because every failure had a named state to fall into instead of a crack to fall through.

15.5. Where the Evidence Lives

All of this talk of "structured evidence" and "durable journals" describes something concrete: four separate stores, cross-referenced only by exact identity, because a factory that keeps all its truth in one place has a single point of amnesia.

PostgreSQL is the control plane and the ledger. It holds each note's lifecycle and, when parked, the structured refusal as JSON; the planning journals; the arm identities; the run ledger, one row per agent run; the refinery's immutable attempts and per-shard gate verdicts; and the promotion's own tables — runs, hosts, effects, events, drains — created by a declared, checksummed migration. Nothing here is a log in the casual sense: rows are written under compare-and-set with explicit version columns, and payloads carry their own hashes.

Dolt holds the task graphs. The Beads databases, one per tool, store the root and child beads and their close reasons — and this is where the binding trick becomes visible: a closed child's close reason is a machine-readable string embedding the note id, the child id, and the exact result-commit SHA. It is a foreign key, in effect, from the task database into Git.

Git holds the artifacts. The branches, the result commits, the deterministic merge candidates. The databases never store code; they store *claims about* code, always as exact SHAs, which the contracts verify against the remote on every resumption.

The filesystem holds the runtime truth. Immutable release trees, root-owned and read-only, each carrying its `RELEASE_SHA` and canonical completion marker; spooled archives beside their digests; and `/proc`, consulted directly for process identity — the involuntary witness no daemon can lie to.

No single store is trusted alone. A completion proof triangulates the Postgres claim against the Git remote; a promotion proof triangulates the pointer on disk against the marker bytes against the running process. **When two stores disagree, that disagreement *is* the refusal.**

15.5.1. One Row, Read Closely

The clearest way to see the ledger work is to read a single row from it. During the campaign, an operator queried the promotion effects table mid-run and found this — an abridged effect 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
```

Read it the way the contracts do. The idempotency key binds this effect to one promotion, one fencing generation, one host, one kind — a crashed worker that restarts finds *this* row rather than repeating the restart (rules 3 and 5). `intent_at` precedes the mutation; the hashed payload records what was *about* to be done (rule 3). Seventeen seconds later `observed_at` records what the world actually said — and the world said no: the daemon's `/proc` executable did not resolve to the release interpreter (rule 1). Cross-referencing that observed time against the daemon's start timestamp showed the proof had sampled `/proc` half a second into the exec; querying the same identity minutes later showed a perfect match. The row did not merely record a failure — it recorded enough to *acquit the host and indict the proof*. The fix cites this row as its reproduction, and the regression test that guards it simulates exactly the slow-exec window the timestamps describe. That is the ledger working as designed: no grepping of scrollback, no reliance on what anyone remembered — the evidence was written down, before and after the act, by the machinery itself, where a successor could query it cold.

15.6. What a Reliability Contract Is

So, finally, the definition the story has been assembling. 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 so every mutation is recoverable and every replay idempotent; re-proves rather than remembering, recomputing from durable state; and binds work to exact identity so "close enough" cannot match.

The campaign's frontier is honest too, and filed: broader provider redundancy so no single credit window stalls the pipeline, typed models at every serialization boundary so shape confusion becomes unrepresentable, batch planning with shared context. Each is a contract waiting to be earned the same way the others were — by a failure, documented.

A companion primer — table-form states, refusal codes, commands, and evidence locations, written to be loaded by humans and agents alike — lives in the documentation repository alongside this book; the full requirement-by-requirement production audit of the campaign is preserved beside it. But the one sentence to carry out of the whole book is the asymmetry: **build the machinery so that its components can fail all day, and its guarantees cannot fail once.**

16 The Future of Yesod

This chapter is deliberately speculative. It describes directions suggested by the current architecture, not features guaranteed by the pinned source.

The most interesting future for Yesod is not "more agents." It is a **learning ecology of tools, processes, evidence, and bounded automation**.

16.1. A Typed Tool Ecology

Today a process step names a tool, action, and notes. A future process language could add typed inputs, outputs, artifacts, side effects, and capability requirements.

For example, a render step could declare that it consumes Markdown plus a theme and emits a PDF with a content hash. A publish step could require a PDF under a size limit and emit a public URL. Yesod could validate the composition before execution and suggest adapters when two tools do not match.

Tool notes would become more than prose around binaries. They could describe capability contracts:

- accepted media types;
- authentication class;
- idempotency;
- expected cost and duration;
- side-effect level;
- rollback or compensation behavior;
- version compatibility.

The registry would then answer, "Which of my tools can satisfy this output contract?" rather than only, "Which tool mentions PDF?"

16.2. Processes as a Workflow Compiler

The molecule model already supports prototypes, DAGs, pours, schedules, and squashes. A natural next step is a compiler from a declarative process to an executable plan.

The compiler could:

1. resolve each capability to a registered tool;
2. validate typed edges;

3. insert conversion steps;
4. estimate cost and critical path;
5. choose local, remote, human, or agent executors;
6. attach acceptance contracts;
7. generate compensation steps for side effects;
8. pour the result as a Beads molecule;
9. preserve a signed execution manifest.

This would make Yesod useful beyond software development: media pipelines, research workflows, publication, data maintenance, and personal operations could share the same durable composition model.

16.3. Reciprocal Tool Improvement

Tools already share a registry and can receive feature requests or bug reports discovered in another tool's workflow. That can become a disciplined reciprocal learning network.

When a process fails at the boundary between two tools, Yesod could propose a structured integration issue containing:

- producer and consumer contracts;
- the concrete artifact that failed;
- the process and step IDs;
- relevant usage notes;
- a suggested owner;
- evidence sufficient to reproduce;
- whether an adapter or source change is cheaper.

Anti-spam and authority rules would be essential. Automatic cross-tool filing should require confidence, deduplicate against active notes, and distinguish observed failure from inferred blame.

16.4. An Adaptive Dispatch Economist

The current system records the pieces of a better router: complexity, project, lane, reason, tokens, cost, outcome, merge, churn, and gate history.

A future router could act as a contextual decision system. It would estimate delivered cost and risk for each eligible lane, choose within a task budget, and reserve controlled exploration. The policy would be project-specific and time-aware: a lane degraded by rate limits today should not inherit its long-term score unchanged.

The hard part is the reward function. It should include:

- accepted merge;
- deployment success;
- absence of short-term revert;

- low corrective churn;
- time to delivery;
- human intervention;
- total attempt and gate cost.

It should not optimize for tokens saved, commits produced, or agent self-reported success.

Every automated route should remain explainable: predicted cost, confidence, rejected alternatives, budget, and policy version.

16.5. A Factory Digital Twin

Yesod's event history could support replay and simulation.

Before changing retry caps, host capacity, gate sharding, or lane policy, an operator could replay recent arrivals through a model of the factory and compare:

- queue dwell time;
- runner utilization;
- merge throughput;
- expected spend;
- blocked work;
- gate fleet size;
- recovery load.

The digital twin would not need to simulate LLM reasoning. It could use empirical distributions from the run ledger and gate history. That is enough to test many operational policies without risking the live fleet.

16.6. A Reliability Immune System

The present factory has local recovery mechanisms but no unified external alert and remediation layer. The next reliability step is not another restart loop. It is a host-aware immune system.

Such a system would:

- collect queue age, disk, schema count, log growth, gate duration, runner drift, and image identity;
- compare them to explicit thresholds and baselines;
- attach a cause fingerprint to each incident;
- suppress repeated alerts only after acknowledgment;
- route host-specific actions to the correct executor;
- cap automatic recovery attempts;
- escalate with a complete evidence bundle;
- verify that remediation changed the fingerprint.

The cause fingerprint solves a current weakness in automatic retry resurrection. A blocked item should reopen because the underlying condition changed, not merely because two hours passed.

16.7. Signed Provenance from Intent to Runtime

The proof chain can become a cryptographic provenance chain.

A future merge could produce an attestation linking:

- source note and bead graph;
- planning snapshot;
- route and policy version;
- run IDs and model identities;
- commits and tests;
- gate image and recipe digest;
- merged SHA;
- deploy artifact;
- runtime version.

This would make Yesod's source-grounded culture compatible with supply-chain standards. It would also make remote gates less mysterious: a verdict would be tied to a known image built from a known commit and recipe.

16.8. Multi-Repository Change Sets

The refinery already supports many repositories, but each owned profile drains independently. Some changes require coordinated releases across a producer, consumer, schema, and deployment.

A future change set could express:

- cross-repository dependency edges;
- compatible version ranges;
- staged gates;
- atomic-or-compensated deployment;
- canary order;
- shared rollback evidence.

This is harder than batching branches in one repository because Git offers no cross-repository transaction. The right model is likely a release molecule with explicit checkpoints and compensations, not a pretend atomic merge.

16.9. First-Class Upstream Contribution

The source already contains an upstream-contribution finish path and profile metadata. Connecting that path safely would let Yesod prepare changes for tools Stephen does not own.

The integration authority changes at the boundary. Yesod can gate and push a fork, but an upstream maintainer owns the final merge. The note lifecycle should therefore distinguish:

- fork branch proved;
- pull request opened;
- upstream review requested;
- upstream merged or declined;
- local consumer updated.

This would turn the catalog's internal/team/external tool kinds into execution policy, not only search metadata.

16.10. Living Documentation

This book itself suggests a future capability: a documentation sentinel that continuously compares claims against source symbols, deployment observations, and schema behavior.

It could flag statements such as:

- a status declared but never written;
- a profile described as planned but present in production;
- a service documented on one host and observed on another;
- a test command that changed;
- a safety control whose tests mock a different API shape from sibling production code.

The output should not auto-rewrite authoritative prose blindly. It should produce a reviewable evidence diff: claim, old source, new source, confidence, and suggested disposition.

16.11. What Yesod Should Not Become

The future also needs negative requirements.

Yesod should not become an opaque swarm whose agents can mutate production because another agent said it was fine. It should not optimize a single metric until every task looks like the metric. It should not let auto-generated notes flood the catalog. It should not erase history to keep dashboards green. It should not allow self-modification to bypass the same gate required of ordinary work.

Most importantly, it should not confuse autonomy with the absence of human authority. A good factory reduces the number of decisions that require attention while making the remaining decisions better informed.

16.12. A Plausible Sequence

The future can be staged.

Near term: much of what earlier editions listed here has since shipped under the reliability redesign—the standing AME was retired in favor of the deterministic controller, the gate proxy is a managed service, deploy/runtime proof landed as the four-host promotion transaction, and the MicroVM fleet count is now snapshotted with its counting contract pinned by a regression test. What remains near-term is narrower: make any residual watchdog actions host-addressed, wire retention cleanup, close the image-provenance gap, and add provider redundancy so no single credit window stalls the pipeline.

Medium term: add cause-aware recovery, connect upstream profiles, type process artifacts, and make cost recommendations project-specific with confidence.

Long term: compile processes across tools and people, simulate the factory, sign provenance, and coordinate multi-repository release molecules.

The order follows the same rule that built Yesod: solve the concrete failure in front of the system, record the evidence, and turn the lesson into mechanism.

Epilogue: Scar Tissue

The impressive part of Yesod is not that it can ask an LLM to edit code. Many systems can do that.

The impressive part 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.

Each refusal is scar tissue from a failure that once looked plausible.

That is what turns an orchestrator into a factory: not the number of agents, but the number of important claims the system knows how to prove.

A Command Field Guide

This appendix emphasizes read-only discovery. Commands that change state are labeled.

A.1. Catalog and Knowledge

```
yesod info
yesod tools list
yesod tool TOOL notes --detail
yesod search "TERM"
yesod query TOOL -q "QUESTION"
yesod ask "QUESTION"
yesod topics list
yesod topic TOPIC notes
```

Create durable knowledge **(writes catalog state)**:

```
yesod tool TOOL usage-note "..."
yesod tool TOOL feature-request "..."
yesod tool TOOL bug-report "..."
yesod tool TOOL note-comment NOTE_ID "..."
yesod tool TOOL supersede NOTE_ID --with "replacement"
```

Use `supersede` when a fact changed and history matters. Use in-place edit only for a typo or equivalent correction.

A.2. Processes and Molecules

```
yesod processes list
yesod processes show PROCESS
yesod processes validate
yesod processes versions PROCESS
yesod processes runs
yesod schedules list
```

Author and instantiate **(writes process/work state)**:

```
yesod processes add PROCESS -d "..." \
  -s 'tool:action:notes' \
  -s 'other-tool:action:notes'

yesod processes add DAG_PROCESS \
  -s 'tool-a:prepare' \
  -s 'tool-b:prepare' \
  -s 'tool-c:combine' \
  --dep 3:1,2

yesod processes sync
yesod processes pour PROCESS
yesod processes squash RUN
```

A.3. Work Graphs

Always route Beads through Yesod for the correct per-tool workspace:

```
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 -- ready --json
```

A.4. Factory Status

```
yesod factory status
yesod fleet agents
yesod fleet status --pretty
yesod agent roster
yesod agent list
yesod agent inspect ADDRESS
yesod daemon-sweep
```

A.5. Dispatch and Runs

```
yesod codefactory-dispatch status
yesod codefactory-dispatch queue
yesod codefactory-dispatch routing-log
yesod codefactory-dispatch logs
yesod runs list
yesod runs show RUN_ID
```

```
yesod runs failures
yesod runs logless
yesod runs rollup ROOT_BEAD_ID
yesod runs planning-report
```

Routing controls **(writes dispatch state)**:

```
yesod codefactory-dispatch arm NOTE_ID LANE --reason "..."
yesod codefactory-dispatch hold NOTE_ID --reason "..."
yesod codefactory-dispatch unhold NOTE_ID LANE --reason "..."
yesod runs kill RUN_ID
```

A.6. Refinery

```
yesod refinery status --pretty
yesod refinery list
yesod refinery show QUEUE_ID
yesod refinery profile list
yesod refinery check-stale
yesod main-ci status
yesod main-ci list
```

Queue mutations such as `dequeue`, `set-order`, `set-status`, and `bypass-blocked` change integration state. Capture evidence and follow operator authority before using them.

A.7. Propulsion

```
yesod sps preview
yesod sps strategy
yesod sps remind
```

`sps preview` generates what would be sent without sending mail, making it useful for inspecting the current findings logic.

B States and Timers

B.1. Note State

State	Operational meaning
`open`	eligible for planning or dispatch when armed and ready
`claimed`	runner owns the note transitionally
`in_progress`	worker attempt is active
`awaiting_verification`	result exists but did not qualify for merge, or is deliberately parked
`awaiting_merge`	branch should be integrated; queue state supplies sub-status
`merging`	declared lifecycle value, not a normal current automated transition
`done`	landed disposition, synchronized to Beads
`blocked`	runner or queue retry/no-progress cap; present outside canonical lifecycle tuple
`wontfix`	terminal human disposition
`superseded`	terminal replacement/duplicate disposition

Usage notes instead use `current`, `stale`, and `superseded`.

B.2. Dispatch State

`agent_dispatch` is orthogonal to lifecycle:

Value	Meaning
`NULL`/empty	unarmed
real lane	armed for eligible runners
`hold`	explicitly paused

Host pins, soft affinity, model plans, retry time, dependencies, and workspace readiness further constrain claimability.

B.3. Run State

The run ledger admits:

`running`, `success`, `failure`, `killed`, `timeout`, `lost`, and `token_budget`.

The run verdict and failure mode add richer semantics; do not infer the whole outcome from this one field.

B.4. Merge Queue State

Declared values are:

`queued`, `gating`, `gate_green`, `gate_red`, `verifying`, `merging`, `blocked`, and `held`.

Not every declared value is an actively used phase in the current automated path. Fine-grained attempt phase belongs in merge-job and worker telemetry.

B.5. Timing Reference

Mechanism	Default / observed contract
Runner poll	10 s
Runner heartbeat	30 s
Dead runner	300 s
Stalled note	300 s
Reaper cadence	60 s
Runtime cap	1,800 s
Kill poll	5 s
Token sample	30 s
Termination grace	30 s
Acceptance timeout	120 s
TUI smoke timeout	600 s
Execution retry cap	3
No-progress early stop	2 consecutive
Execution backoff	`2^retry` minutes
Soft affinity age-out	300 s
AME supervisor poll	30 s
Queue gate-red retry cap	2
Stale blocked inspection floor	10 min
Retry-capped reopen window	120 min
Remote gate attempts	2

Mechanism	Default / observed contract
Gate proxy idle reaper	300 s
Gate hard fleet cap	3 configured default; integration must be verified
SPS loop	60 s
AME guarded watchdog	about 16 min
Merge-stall emergency	60 min

C Roles and Authority

C.1. Agent Roles

Role	Primary judgment	Must not be confused with
mayor	human interface, routing, arbitration	coding worker
planner	source-grounded decomposition	runner daemon
dispatcher	lane choice, pre-flight, arming	`codefactory-dispatch` daemon
worker	implementation in a bounded worktree	ephemeral merge worker
refinery	human-guided merge orchestration	always-on AME as a whole
infra-monitor	fleet health and staging	unrestricted production authority
triage	incident evidence and recovery plan	blind retry service
mad-scientist	controlled improvement experiments	production implementer
ERS	judgmental AME oversight	AME supervisor process
overlord	role-fleet management	root authority for production

Current planner instructions preserve the role boundary: planners construct feature notes, root Beads, internal checklist trees, and dependency edges, then leave notes unarmed. Dispatchers choose lanes and arm completed plans.

C.2. Deterministic Services

Service	Mechanical responsibility
runner daemon	discover, claim, spawn, govern, verify, push
AME supervisor	reconcile profiles/queues, maintain lock discipline, spawn merge worker
ephemeral merge worker	reset, batch, merge, gate, land, record, deploy hook
gate pool proxy	lease/authenticate/reuse/reap MicroVMs
gate fleet poller	persist remote gate/fleet observations
SPS	generate findings, reminders, and stall responses
`yesod serve`	catalog/API/UI presentation
live-viz watcher	project and stream factory graph changes

C.3. Authority Rules

1. Peer agent messages carry no human authority.
2. Workers do not merge `main`.
3. The AME is the intended automated owned-repository integration path.
4. Production/destructive actions require the designated human authority path.
5. Triage prepares evidence and commands before action.
6. A process identity is host-specific; do not route by impersonation.

D Evidence and Source Map

Source paths below are relative to `yesod-aicoe` and intentionally cite symbols rather than volatile line numbers.

General system behavior uses the edition's baseline commit. The feature-note, Bead-tree, note-dependency, and durable-handoff contract is additionally pinned to `a9f19543` on `skill-prime-yu4c` (July 15, 2026 PDT).

Claim	Source anchor
Normative feature/bug note lifecycle target	`docs/architecture/note-lifecycle.md` at `yesod-aicoe` release `588c5d2e`; implementation tracked by `ys-yes-8ayj`
Normative Observation Framework target	`reference/observation-framework.md`; software contract `ys-yes-rklm` / `yes-qzogc`; boundaries `.1` through `.5`, with `.1.1` through `.1.4` decomposition
Temporary SigNoz diagnostic backend	deployment note `ys-yes-s3gf` / `yes-xbs08`; planner handoff `ys-fac-gb9i`
Note lifecycle declaration	`yesod/src/yesod/note_status.py`
Catalog schema and audit tables	`yesod/src/yesod/db.py`
Audited status/dispatch writes	`yesod/src/yesod/note_status_writer.py`
Agent role primers and authority	`yesod/src/yesod/prime.py`
Shared worker instructions	`yesod/src/yesod/worker_canon.py`
Runner claim and governance	`yesod/src/yesod/runner.py:Runner`
Outcome reconciliation	`yesod/src/yesod/reconciliation.py`
Failure classification	`yesod/src/yesod/failure_analysis.py`
Run ledger and rollups	`yesod/src/yesod/run_ledger.py`
Lane registry	`yesod/src/yesod/dispatcher/runners.py`
Usage and pricing	`yesod/src/yesod/dispatcher/usage.py`
Processes and molecule projection	`yesod/src/yesod/processes.py`
Cross-catalog question answering	`yesod/src/yesod/ask.py`, `query.py`
Queue reconciliation	`yesod/src/yesod/refinery_queue.py`
AME supervisor	`yesod/src/yesod/refinery_supervisor.py`
Ephemeral merge worker	`yesod/src/yesod/ephemeral_worker.py`
Batch/deploy decisions	`yesod/src/yesod/merge_core.py`
DB-backed profiles	`yesod/src/yesod/refinery_profiles.py`

Claim	Source anchor
Post-merge hook	`yesod/src/yesod/post_merge_hook.py`
Gate telemetry	`yesod/src/yesod/gate_telemetry.py`
MicroVM proxy	`yesod/docker/microvm/gate-pool-proxy.py`
Cost/quality statistics	`yesod/src/yesod/stats_data.py`
Propulsion and watchdogs	`yesod/src/yesod/propulsion_service.py`
Queue janitor	`yesod/src/yesod/refinery_janitor.py`
Retention library	`yesod/src/yesod/factory_janitor.py`

D.1. Current Deployment Snapshot

Observed July 11, 2026 PDT:

Plane	Placement
operator/integration	`pompom`
presentation/runtime	`dertog`
catalog	dedicated PostgreSQL 17 VM
work graphs	dedicated Dolt SQL VM
Linux execution	dispatch host plus three runner VMs
cloud gate	disposable MicroVM behind loopback proxy
hypervisor	`seykhl` for local Linux VMs

Host placement is deployment evidence, not a guarantee encoded by the Python modules.

D.2. Documentation Corrections Incorporated Here

This edition intentionally corrects several claims in the earlier reference set:

- multi-repository refinery profiles are live, not merely planned;
- runtime compatibility for exact child notes still exists, but current planner guidance uses one feature note and root Bead as the merge unit;
- planners leave completed feature notes unarmed; dispatchers own lane choice and arming;
- Bead dependencies order work inside one feature, while note dependencies order features across merge boundaries;
- arming has several transports, not only one HTTP endpoint;
- the automated note path does not normally write `merging`;
- status advance and queue enrollment are separate durable operations;
- optional skipped checks are not explicit failures for ordinary code tasks;

- successful worktrees are cleaned; failed/unverified worktrees remain;
- merged remote branches are not automatically deleted by the current path;
- deploy is profile-driven and may skip or fail after merge;
- remote red isolation is sequential by default;
- July 11 storm controls include loopback refusal, failed-launch termination, explicit shutdown, corrected lock ownership, and a hard cap;
- SPS now has guarded and emergency recovery layers, though live cross-host placement undermines local PID signaling;
- janitorial capability is layered: active queue maintenance, shadow refinery janitor, and an unwired retention library;
- threshold coverage remains incomplete despite richer SPS findings;
- the MicroVM hard-cap response-key integration was subsequently verified in live service during the self-hosting exit wave.

E Yesod Development Waves

Yesod changes in bounded development waves. Each wave has a purpose, repository baseline, stopping condition, and evidence-backed result. This chapter is the book's durable wave ledger.

Append each wave as another second-level section. Begin with purpose, window, commits, merges, contributors, change surface, gates, and exit state; then explain why the wave existed, what changed, what stayed out, and what matters next.

E.1. Wave 1: The Self-Hosting Exit

E.1.1. Wave Summary

Field	Value
purpose	complete bounded hardening, retire optional self-work, and pivot to external delivery
development window	July 13, 15:21 to July 14, 17:15 PDT—25 hours 54 minutes
baseline	`557c6b28`
final commit	`b838cd56`
Git commits landed	100 on `main` across the wave range
recorded refinery merges	36 merge summaries
contributors	5 author identities
change surface	80 files, 9,936 insertions, 523 deletions
final gates	2 full exit gates green; zero failures
exit state	4 current runners; queue and gap 0; health 24/24; provisioning 108/108; 0 nonterminal Yesod notes

The commit and merge counts measure different things. The commit count is the Git ancestry introduced between the baseline and final commit; one refinery merge can carry several com-

mits. The merge count comes from Yesod's durable `merge_summaries` ledger and represents recorded integration events.

Every self-hosting system faces a dangerous moment. The machinery becomes interesting enough that improving the machinery feels like progress even when it delays the work the machinery was built to produce.

Yesod reached that moment in July 2026. The factory could plan, dispatch, isolate, gate, merge, deploy, and record its own changes. It also had a long tail of plausible improvements. The answer was not another open-ended hardening program. It was a **locked exit wave**: finish the agreed safety and operability batch, retire or consolidate the rest, and change the success metric from "Yesod improved Yesod" to "Yesod delivered another tool."

This first wave entry exists because the interesting result was not one feature. It was the point at which a collection of mechanisms became credible as a production path for external projects.

E.1.2. The Cutoff Was a Feature

The wave began by drawing a boundary around the work. Existing bug reports and feature requests were classified into four groups:

1. changes required for safe throughput;
2. already-shipped or stale records that needed truthful closure;
3. broader reliability ideas to preserve as roadmap knowledge;
4. optional surfaces to retire explicitly rather than leave ambiguously open.

The distinction matters. An autonomous factory with an unlimited mandate to improve itself never finishes. A factory with a durable cutoff can distinguish a blocker from an attractive idea. During this wave, only faults that blocked the exit batch—or tiny observability changes needed to understand that batch—were allowed to expand the plan.

That governance became executable, not merely conversational. Yesod gained an orthogonal disposition axis—active, held, blocked, needs-rework, quarantined, or deferred—separate from lifecycle status. A task could remain `awaiting_merge` without being eligible to gate. Releasing a previously held task became an explicit audited action. This prevented the familiar failure in which a dashboard said "waiting" while a hidden policy said "never run."

E.1.3. What Shipped

The following table groups the most important outcomes. Note identifiers are included because the catalog is the durable narrative; commits are included because the repository is the executable evidence.

Area	Result	Evidence
batch safety	conflicted and cross-tool branches fail closed and cannot contaminate a shared gate tree	`ys-yes-gpgf`, `ab4384af`
queue governance	held work stays out of play; retry caps, dependency state, and needs-rework are durable rather than sentinel conventions	`ys-yes-d9yc`, `283d3e71`
external proof	a real `clip-together` change traversed the automated merge lane successfully	`ys-cli-eij7`, `489f9f0f`
persistent evidence	blob storage received guarded, persistent mount configuration rather than relying on a process-local filesystem	`ys-yes-e3ab`, `f34d45cf`
gate cleanup	proxy shutdown and failure paths terminate tracked instances and sweep orphans instead of leaking cloud capacity	`ys-yes-ph61`, `947bfec0`
tailnet enrollment	gate guests join Tailscale during readiness using an ephemeral key fetched from Secrets Manager, with no key baked into the image	`ys-yes-bwoz`, `ec68fe5d`
least privilege	the gate can reach the telemetry collector without receiving general tailnet authority; the image includes an explicit security probe	`ys-yes-bwoz`, `ec68fe5d`
live telemetry	structured gate progress reaches the collector, and the Refineries view shows warming, running, progress, failure, and terminal green state	`ys-yes-x8yf`, `ab56e8b8`

Area	Result	Evidence
managed Beads truth	`yesod bd … backend` uses the same resolved Dolt host, user, password, port, and TLS policy as routed execution without printing the password	`ys-yes-6ew3`, `075df362`
cold-start reliability	`/ready` treats HTTP 503 as "still baking" for a bounded 180-second window while other HTTP and transport failures remain fail-fast	`ys-yes-bv5g`, `b838cd56`

Several of these changes are small in diff size and large in operational effect. Passing `executionRoleArn` through the proxy fixed a cloud launch path that appeared idle from the control plane. Selecting the newest image by creation time prevented a lexicographic version mistake. Configuring Git identity inside the gate turned a baffling merge failure into an ordinary, testable precondition. Preserving terminal status after shutdown let an operator distinguish "nothing happened" from "the gate finished and the capacity was released."

E.1.4. A MicroVM Became a Managed Gate

Before the wave, the remote gate was best understood as a powerful execution endpoint with several dangerous edge cases. After the wave, it behaved more like a managed subsystem.

The host-side client can reach only a loopback pool proxy, not an arbitrary cloud runtime URL. The proxy owns launch serialization, one tracked instance, idle shutdown, explicit shutdown, failed-launch cleanup, and orphan sweeping. The guest receives runtime configuration at `/ready`, fetches short-lived credentials, joins the tailnet in userspace, initializes its repository and database, and reports structured progress. The collector receives evidence; the guest does not receive broad control-plane authority.

The final cold-start defect is a good example of why all these layers matter. A cold guest legitimately returned 503 while building its environment. The worker interpreted two immediate 503 responses as two failed gates even though no tests had run. The fix did not raise every retry count or make all errors slower. It introduced typed HTTP status on the gate error and retried only the expected readiness response, every five seconds, within a 180-second budget. Transport errors and HTTP 500 still fail immediately. Whole-gate attempts remain capped.

That is the desired pattern: identify the state that is actually transitional, give it a bounded budget, and keep every other failure sharp.

E.1.5. Evidence at the Exit

The wave closed with live evidence, not a declaration of confidence.

- Thirty-six changes entered recorded merge summaries between the start of the locked evening batch and the final merge.
- The managed-Dolt diagnostic reported `managed:doltsvr:3306`, reachable, on Dolt `2.1.10`; the exact JSON acceptance probe exited successfully.
- The final backend-diagnostic gate collected 4,636 tests, completed 4,614, skipped 22, reported zero failures, and returned green.
- The final cold-readiness gate collected 4,639 tests, completed 4,617, skipped 22, reported zero failures, and returned green.
- A post-merge live canary deliberately removed the warm instance, launched a fresh MicroVM, and reached `/ready` in 68.5 seconds through the deployed client. The canary then shut the instance down.
- Four dispatch runners were active with no claimed work. The refinery was active with an empty queue, a free merge lock, and a zero queue gap.
- PostgreSQL, every registered Dolt workspace, and the full 108-cell fleet-provisioning matrix reported healthy.
- The operator checkout, editable runtime checkout, AME checkout, and all four dispatch runner runtimes resolved to `b838cd56` after rollout; host-local runner configuration and backups were preserved across the fast-forward.

At the code exit, one old note remained in `awaiting_verification`: `ys-yes-5mhm`, a mail-broker remap experiment explicitly excluded from the locked final scope. Final disposition cleanup marked it `wontfix` and closed its Bead without merging. Its tested branch remains preserved at `6adc3e95` so future external-project evidence can justify a deliberate refile rather than silently resurrecting old self-work.

E.1.6. Ready Does Not Mean Configuration-Free

The factory is ready to work on an external project, but an external repository still needs a contract. Yesod must know where the repository and Beads workspace live, which lanes may execute it, which branch namespace the factory owns, what command constitutes a green gate, and whether a merge triggers a deployment. Secrets must be available on the hosts that need them without being copied into prompts or images.

The readiness claim is therefore specific. `clip-together` has recent end-to-end proof through dispatch, gate, refinery, and `main`. Several other owned tools already have complete refinery profiles. An arbitrary catalog tool is not automatically merge-ready: some have no profile or dedicated refinery checkout, and Git write authority is repository-specific. Before the first task, both a runner and the refinery host should prove remote access with `ls-remote` and a non-mutating push dry run. The foreign-tool guards fail closed against merging in the wrong repository, but missing configuration can still strand otherwise valid work.

The first task on a new project should therefore be a canary: small, reversible, representative of the real test toolchain, and valuable enough to merge. It should prove the complete path—note, Bead, role priming, branch, targeted test, remote push, refinery profile, gate, merge, and post-merge observation—before parallelism is increased.

This is not hesitation. It is how a factory converts a new project from an assumption into a known production line.

E.1.7. The New Metric

The exit criterion changes what Yesod should optimize.

Future maintenance is justified when external work exposes a concrete fault, when operating evidence shows a safety boundary is weak, or when a small change materially increases throughput across projects. It is not justified merely because Yesod can imagine another abstraction for itself.

The factory's next meaningful chart is therefore not the number of Yesod notes closed. It is external lead time: how quickly a well-formed request becomes a tested, merged, observable change in another tool—and how often that happens without extraordinary operator intervention.

E.2. Wave 2: The Async Gate and the Operator Surface

E.2.1. Wave Summary

Field	Value
purpose	turn the remote gate into an asynchronous, shard-capable job protocol; convert incident lessons into safe operator commands; make the control plane honest about local and remote execution
development window	July 15, 11:13 to July 20, 20:18 PDT—5 days 9 hours 5 minutes
baseline	`b838cd56`
final commit	`3ca2edff`
Git commits landed	77 total: 65 non-merge commits and 12 merge commits
contributors	4 Git author identities
change surface	81 files, 10,280 insertions, 613 deletions
completed feature notes	12 feature requests created after the Wave 1 cutoff, plus incident-driven bug fixes
exit state	async and shard code merged; operator CLI gaps closed; 22 refinery profiles normally gate locally; MicroVM cutover rolled back after a broken image; observability work remains active

Wave 1 ended with a production path that was safe enough to deliver external work. Wave 2 began when that path met reality. Between July 15 and July 17, operators repeatedly reached

for direct database queries during recovery. At the same time, the remote gate still held a long-running request open, and the first attempt to make it asynchronous exposed a deeper image and deployment problem.

The wave therefore had two connected objectives. The first was architectural: replace a long, blocking gate request with a job protocol that could support parallel shards. The second was operational: make the state, permissions, and repair actions visible through Yesod itself. The resulting wave is not a story of a clean remote cutover. It is a story of a better protocol, better control surfaces, and a sharper boundary around what had actually been proven.

E.2.2. The Gate Became a Job Protocol

The first three steps of the remote-gate chain changed the unit of work:

1. `POST /run` validates the request, writes a queued job record, starts the gate in a background thread, and returns a job identifier quickly.
2. `GET /status/<job_id>` reads the job record and progress files without waiting for pytest. It reports phases such as `queued`, `fetching`, `running`, `done`, and `error`.
3. The final verdict is written before it is reported. A client that loses its connection can recover the result by polling the durable job record.

The lock discipline matters as much as the endpoint shape. The worker holds the setup lock for database and batch preparation, but pytest runs without holding that lock, so status polling does not compete with the whole test run. This turns a request timeout from an ambiguous failure into a recoverable observation problem.

The protocol then grew a second dimension. A run can carry `shard_index` and `shard_count`; `pytest-split` partitions the suite; and the pool proxy launches and tracks a fleet rather than one MicroVM. The proxy polls each job, reports per-shard progress, retries bounded infrastructure failures, and computes the overall verdict as the conjunction of the shard verdicts. A dead or setup-failed shard is infrastructure failure, not a red test result.

This is a meaningful change in the factory's state model. The gate is no longer "one HTTP request that eventually returns a verdict." It is a collection of durable jobs whose progress, failure class, and terminal evidence can be reconciled independently.

The supporting work followed the same logic. The image build gained optional duration data for shard balancing and a flavor-specific provisioning path. Launch and reap loops gained jitter so several workers do not synchronize expensive AWS polls. Agent spawning gained asynchronous prompt delivery and session pruning after an incident left more than 160 stale opencode sessions. The intended performance target was a shard fleet completing in roughly four to eight minutes instead of a roughly fifty-minute local gate. That target motivated the design; it was not established as a production guarantee by the end of this wave.

E.2.3. The Cutover Failed at the Image Boundary

The most important evidence in this wave is the failed remote cutover.

The rebuilt `yesod-gate-async-shard` image, active as version 26.0, carried a venv without the `yesod` package. Every shard reached setup, attempted to load the pytest plugin, and failed with `ModuleNotFoundError: No module named yesod`; zero tests ran. A gate attempt burned sixteen MicroVMs before the storm guard stopped the launch pattern. The operator rolled the supervisor back to `YESOD_GATE_BACKEND=local`, exactly as the rollback plan required.

The root cause was more specific than "the image was bad." The general flavor still contained a TTRAC repository URL and gate database name inherited from an earlier lineage. Because `build.sh` was the input for the Yesod async-shard image, the image could be structurally healthy while cloning and configuring the wrong project. The follow-up added a build-time image-environment self-check and replaced the stale TTRAC values in the Yesod path. Those fixes improve the next build, but they do not retroactively make the failed image a green end-to-end proof.

The operating state at the close of the wave was therefore explicit: all 22 refinery profiles were gating locally. Local execution was the normal path, not a degraded remote fallback. The MicroVM implementation remained present and was rendered accurately in the UI for a later, operator-gated cutover. This distinction prevented a fast protocol from being mistaken for a proven deployment.

E.2.4. The Operator Stopped Needing `psql`

The incident marathon also produced the most practical feature cluster in the wave. A durable operator surface replaced roughly 25 direct database pokes, including 10 writes:

Area	New capability	Evidence
note administration	inspect status, disposition, attempts, branch, and dispatch together; set a validated branch; reset gate attempts with an audit reason	`ys-yes-00fd`, `fb293fc0`
refinery administration	inspect/edit profiles, inspect the merge lock, list workers, and reap only verified-dead rows	`ys-yes-00fd`, `4a4c7184`
truthful views	compute `branch_exists` from the profile's repository rather than the caller's current directory; expose queue state and age	`ys-yes-00fd`, `4a4c7184`
diagnostics	answer "why is this armed note not running?" and check test-schema health from sanctioned commands	`ys-yes-00fd`, `1a986359`

Area	New capability	Evidence
mail	read complete message bodies from scripts; purge off-roster agent mail while preserving human channels	`ys-yes-00fd`, `ys-yes-gchl`
recovery	rearm a repaired refinery item atomically instead of manually resetting monotonic counters	`ys-yes-lgsz`, `fadf1483`

The important result is not command count. It is that repair actions now have names, validation, and an audit trail. A branch repair can verify that the remote ref exists. A gate-attempt reset can carry a reason. A worker reap can require evidence that the process is really dead. The control plane is moving from "the operator knows which tables to edit" to "the system exposes the allowed state transitions."

That is also why the wave added explicit model-and-effort lanes, tool-scoped gating marks, and a visible `planning` status. Model selection, work ownership, and planning are policy decisions; they should not be hidden in free-form prompts or inferred from whichever worker happens to claim a row. The guidance change that one note represents one independently mergeable feature, while its Bead tree represents internal sequencing, made this boundary explicit (`ys-yes-yu4c`, `712d5c52`).

E.2.5. Observability Learned to Name States

The Refineries page had been shaped around the MicroVM path even while local gating was the live production state. The wave made it backend-aware: local and MicroVM gates now have distinct rendering and progress helpers, and the page retains remote detail without implying that remote execution is active. The final readability pass also removed a stale panel timestamp, labeled gate speed as relative to supervisor start after a restart, and replaced a bare zero-percent collection bar with an explicit `collecting` state.

Those changes reflect a broader lesson. A percentage is not a state machine. During pytest collection, zero percent can be healthy. After a supervisor restart, "no completed gate" can mean "no completed gate in this generation," not "no gate has completed." A dashboard that collapses these distinctions creates operational work even when the underlying worker is correct.

The last review of this wave also found two gaps that the final clarity pass did not yet close. Local gates did not export the same progress signal as the fleet path, so a healthy long-running local gate could still appear stalled. And the worker could stop heartbeating while blocked in pytest, allowing a live worker to look dead after the ghost threshold. Those findings belong in the next wave, not in the shipped-results column.

E.2.6. What Shipped, and What Stayed Out

Area	Shipped result	Boundary at wave exit
async execution	job identifiers, background execution, status polling, write-ahead verdicts	full remote production cutover was not proven
shard execution	pytest-split integration, pool-proxy fan-out, per-shard aggregation and bounded infra retry	the first rebuilt image failed before tests ran
image safety	Yesod-specific environment correction and build-time image self-checks	one green end-to-end rebuild and cutover approval still required
operator control	sanctioned note, refinery, mail, health, and recovery commands	direct SQL knowledge remains useful for diagnosis, but is no longer the normal repair interface
dispatch policy	explicit Codex model/effort lanes, planning lifecycle adoption, tool-scoped gating	model economics still need observation across external work
observability	backend-aware rendering and clearer collection/timestamp semantics	local progress export and heartbeat semantics remain open

The evidence is therefore layered. Seventy-seven Git commits landed across 81 files, and feature-specific tests accompanied the async, sharding, CLI, and observability changes. The remote image incident supplied a stronger kind of evidence than a green unit suite: it showed that build inputs, package contents, runtime environment, and rollback behavior all belong to the gate contract. A focused validation run covering the async gate server, pool proxy, refinery CLI, CLI black-box, and backend-aware Refineries tests passed 304 tests. The local rollback succeeded; the remote cutover did not.

E.2.7. The New Exit Criterion

Wave 1 asked whether Yesod could safely leave self-hosting mode. Wave 2 asks a more demanding question: can a distributed execution design be proved at the same boundary where it will operate?

The next cutover should require a build-time environment check, one single instance `/run` diagnostic, one genuinely green shard, and then one complete multi-shard gate whose verdict matches a local run. Only after that evidence should the supervisor be pointed at the fleet. Separately, the local path needs progress and heartbeat semantics that cannot declare a healthy gate stalled merely because pytest is quiet.

The wave's durable lesson is simple: asynchronous execution increases the need for explicit state; parallel execution increases the need for typed failure; and operational autonomy in-

creases the need for sanctioned, audited control surfaces. Speed is valuable, but the factory earns the right to use that speed only when the image, worker, gate, and dashboard tell the same story.

E.3. Wave 3: The Crash-Only Factory

E.3.1. Wave Summary

Field	Value
purpose	turn the asynchronous gate into an operationally safe fleet; make every refinery, repository, and agent lane explicit; remove hidden fallback and liveness assumptions
development window	July 20, 13:55 to July 26, 13:37 PDT—5 days 23 hours 43 minutes
baseline	`3ca2edff`
final commit	`31938769`
Git commits landed	94 total: 83 non-merge commits and 11 merge commits
contributors	4 Git author identities
change surface	96 files, 13,670 insertions, 735 deletions
exit evidence	93 focused fleet tests passed; 12+ complete eight-shard fleets launched and reaped cleanly; no `ThrottlingException` in roughly 870 subsequent proxy-log lines
exit state	Yesod's gate defaults to MicroVM with explicit local rollback; refinery recovery is lease- and checkpoint-based; per-tool routing and foreign-repository credentials are explicit; the project and agent factory can provision more than one kind of lane

Wave 2 built the asynchronous gate protocol and ended with a failed image cutover. Wave 3 dealt with the consequences of making that protocol real. A remote gate cannot be treated as one more transport implementation: it has leases, fleets, credentials, flavors, repository identity, and failure modes that must agree across the worker, proxy, guest, supervisor, and dashboard.

The central change was a shift from **fallback-oriented operation** to **explicit ownership**. Yesod no longer quietly runs a local gate when the remote path is unavailable. A profile says which repository it gates, which proxy and image it uses, and which credentials it may receive. If that path cannot run, the item remains infrastructure-failed and queued for recovery. The

operator may still select local gating, but that is now a deliberate rollback decision rather than an accidental safety net.

E.3.2. Readiness Became a Control-Plane Fact

The wave opened by fixing a quieter source of starvation: readiness was being reconstructed from several imperfect views. The collector now mirrors Dolt's `ready_issues` into PostgreSQL; the runner claim path reads that mirror and performs a single candidate verification; and a slower reconcile pass compares the mirror with Dolt and alerts on drift (`0547a214` through `95663048`).

That work exposed the default 100-issue limit in `bd ready`. With more than 100 ready Beads, a valid armed note could exist in PostgreSQL and still be invisible to every runner. The runner and collector now request the complete set with `--limit 0`, and a regression test covers a note beyond position 100 (`88c7df18`, `94e24499`, `d539227a`). Readiness is no longer "whatever the last subprocess happened to print."

The same boundary became louder when external commands misbehaved. Yesod now detects and logs trailing stdout pollution instead of silently accepting a partial or contaminated JSON document. Profile-loading exceptions are surfaced, and repeated `gate_infra_unavailable` outcomes escalate through the janitor. These are small changes with a common purpose: a control-plane claim must carry enough evidence to be trusted, and an infrastructure failure must remain distinguishable from a test failure.

E.3.3. The Refinery Became Crash-Only

The largest cluster was the crash-only refinery work. A killed worker should not strand the merge queue, a dead process should not retain the main lock, and a gate whose client connection disappears should not lose its verdict.

The new lifecycle is built from several cooperating mechanisms:

Failure boundary	Mechanism	Evidence
gate result	write the verdict and log checkpoint before reporting completion, so a severed client can recover the result	`cff9186f`, migration `0035`
operator cancellation	`refinery abort-gate` marks the queue item aborted without pretending that it was a test red; gate-attempt accounting remains explicit	`124a3d33`, `31d39168`, migrations `0036`

Failure boundary	Mechanism	Evidence
merge lock	renew a committed lease with heartbeat, record owner PID and host, and reap a dead or stale owner	`eccb266e`, migration `0037`
worker death	a lease-heartbeat thread and abort flag let the worker terminate itself when its ownership is lost	`b85f9fdc`
supervisor hang	bound database connection, statement, and lock waits; abort a tick that exceeds its wall-clock budget	`bebedcd6`, `f7d3af33`, `605f109e`
fleet churn	tear down terminal fleets immediately, exclude terminated ghosts from capacity, and debounce relaunches	`1d0f4367`, `3c086616`, `c0b8323c`

This is stronger than adding retries. Retries are useful only when the system knows who owns the work and which result is authoritative. The lease and checkpoint model makes recovery a state transition: a new worker can see that the old owner is gone, reclaim the lock, inspect the last durable gate state, and continue without guessing whether the previous attempt merged, failed, or was merely disconnected.

The practical trigger was a real AME hang. A supervisor sat in an unbounded Postgres poll for roughly thirty minutes while work waited and the merge lock was free. The fix made database operations bounded and added a crash-only tick watchdog. A separate dashboard heartbeat fix writes at tick start and periodically, and conditions stale/dead display on a live worker row. The dashboard can now distinguish “the supervisor is busy” from “the supervisor is gone.”

E.3.4. The Fleet Survived Its First Real Storm

The first implementation of shard fan-out could create a new failure loop. The gate proxy and the dashboard poller both called AWS `ListMicrovms`; under throttling, a missing count was interpreted too optimistically, shards cycled from running to terminated, and relaunches consumed the fleet cap without reaching pytest.

The fix combined adaptive SDK retries, a TTL cache for fleet listings, terminated-ghost exclusion, partial-fleet adoption and relaunch, two-strike shard-death debounce, and a corrected poller port/interval (`1d0f4367`, with the deployed patch also recorded as `43d18702`). The fleet now tears down terminal instances promptly and retains per-shard failure tails in the gate log (`861d2f39`, `a291d9e6`). A stale fleet snapshot cannot override live `merge_jobs` or `gate_runs` telemetry (`bd283ef1`, `02c276ed`, `3e7b065f`).

The evidence is unusually concrete. The focused proxy, poller, and plist tests passed 93 tests. The subsequent operational note reports more than twelve complete eight-shard fleets launched and reaped cleanly, with zero throttling errors across approximately 870 proxy-log lines; before the fix, the same class of log accumulated roughly 290 throttling errors per hour. This does not prove that every possible repository flavor is green, but it proves that the fleet lifecycle itself stopped self-amplifying under normal eight-shard load.

E.3.5. A Generic Gate Acquired a Repository Contract

Wave 2 showed that a generic image could be asynchronous and still be wrong for its repository. Wave 3 made repository identity a first-class gate input.

The `gate_registry` is now the source of truth for per-tool proxy ports and MicroVM flavors. The live convention reserves `8099` for Yesod, `8100` for TTRAC, and `8101` for SJBGTD; new lanes allocate upward. Python/uv/Postgres and Node flavors are named explicitly, and the `refinery gate-plan` command shows the intended cutover state before an operator changes it (`22bcbea9`, `82c97a35`).

The worker no longer assumes that every remote batch belongs to the warm Yesod repository. Non-Yesod lanes carry their own repository URL, and the worker refuses a foreign lane whose manifest information is missing rather than silently gating the wrong checkout (`3ccefebc`, `6c37b383`). The remote proxy fetches the selected tool's deploy-key secret and injects only the material needed for that job; the guest consumes it during the per-job clone (`f2a803a1`, `9c8d9c94`, `8d2bf045`). The old single baked deploy key was the wrong abstraction for a multi-repository factory.

The yesod profile now defaults to the MicroVM gate, and the automatic local fallback is retired (`cc6374b9`). A remote failure is a typed infrastructure failure that leaves work queued. `YESOD_GATE_BACKEND=local` remains available as an explicit operator rollback. Other languages or bespoke shell stacks without a registered flavor can remain local until their gate image exists; "MicroVM everywhere" became a registry and validation problem, not a blanket assumption.

E.3.6. The Factory Learned to Create Projects and Agents

The wave widened the boundary of Yesod itself. `yesod project new` now scaffolds a new repository from a template, with Rust, Python/uv, and Apple flavors in the initial surface. It initializes the project, creates and shares the GitHub repository, accepts the bot invitation, and registers the project with the factory's workspace, refinery, and Beads conventions (`76402192`, `056f0f18`). The factory is beginning to produce new production lines, not merely changes to its own machinery.

Agent execution received the same treatment. The six-layer spawn repair adds the correct launchd PATH, session inspection through the opencode HTTP API with CLI fallback, spawn preflight with `PATH/YESOD_BIN` injection, prompt delivery checks, a post-spawn registration/liveness gate, and a turn watchdog that re-prompts sessions left with a dangling tool (`32a8f1fc` through `db914780`). This turns "the agent died silently" from an operator intuition into a sequence of checkable lifecycle states.

Dispatch policy also became more mechanical. Arm paths refuse notes that are not open or are already held in merge work unless `--force` is explicit; propulsion excludes done and awaiting-merge notes from false readiness findings; and repeated nudges are rate-limited to roughly two hours instead of flooding the mayor inbox (`e9435a54`). The dead `opencode-claude` lane was removed, an `opencode-minimax` lane was added, and the infra-monitor seat default was provider-prefixed (`25cee8cb`, `31938769`).

E.3.7. The Dashboard Became a Deployable Surface

The dashboard is now aware that it may live behind a reverse proxy rather than at `/`. A request-scoped prefix helper rewrites HTML links, forms, fetches, EventSource URLs, and redirects. Live Viz derives its API base from the pathname and uses secure WebSockets behind HTTPS. The canonical operator URL is now advertised as the proxied Yesod path, while direct local access remains unchanged (`0248d393`, `05da3ae5`, `ba9af467`).

The operator experience also gained state-transition audio controls and a shared LCARS-style audio engine. That work is cosmetic compared with leases and credentials, but it belongs to the same pattern: the factory's state is becoming something an operator can perceive directly, not a collection of database rows and raw logs.

E.3.8. What Shipped, and What Stayed Out

Area	Shipped result	Boundary at wave exit
readiness	PostgreSQL mirror, complete ready-set retrieval, single-candidate verification, and drift alerts	Dolt remains the external readiness source; the mirror must continue reconciling
refinery safety	write-ahead checkpoints, abort-gate, leases, heartbeats, dead-owner reaping, bounded DB calls, and tick watchdog	crash diagnostics and broader supervisor self-healing remain follow-on work
fleet execution	adaptive throttling defense, ghost suppression, clean terminal teardown, per-shard failure retention, and remote-default Yesod gating	non-Python or bespoke flavors still require explicit local treatment
repository isolation	gate registry, per-tool ports/flavors, manifest guards, and per-job deploy-key injection	each new lane still needs operator-approved provisioning and one real end-to-end gate

Area	Shipped result	Boundary at wave exit
factory expansion	project templates, GitHub bot acceptance, six-layer agent spawn health, and safer dispatch nudges	AWS-hosted Yesod remains an exploratory roadmap, not a shipped deployment
operator surface	reverse-proxy-safe dashboard, secure WebSockets, and clearer live state	dashboard polish does not replace the durable database and lease model

E.3.9. The New Exit Criterion

Wave 2 asked whether an asynchronous gate could be made real. Wave 3 asked whether it could be trusted after the process, host, repository, credential, or network around it failed. The answer is now much stronger: the fleet has a recoverable lifecycle, the Yesod lane has an explicit remote default, and a foreign repository cannot silently inherit Yesod's identity or key.

The next wave should measure the factory by lane onboarding and recovery time: how quickly a new repository can be assigned a valid flavor, credential, proxy, refinery profile, and green canary—and how quickly a killed worker or stale lease returns that lane to useful work. The architectural lesson of this wave is that autonomy is not the absence of intervention. It is the presence of enough typed state, bounded waiting, and explicit ownership that intervention becomes rare, safe, and explainable.

E.4. Wave 4: Integration Branches and Bounded Intelligence

E.4.1. Wave Summary

Field	Value
purpose	let multi-note epics accumulate safely away from `main`; bound LLM maintenance cost; close the last reverse-proxy and stall-detection gaps
integration window	July 26, 14:53 to 22:54 PDT—8 hours 1 minute
baseline	`31938769`
final commit	`4505cfbc`

Field	Value
Git commits landed	15 total: 12 non-merge commits and 3 merge commits
contributors	4 Git author identities
change surface	37 files, 2,101 insertions, 285 deletions
principal feature	`branch:<name>` note tags with queue-snapshotted target branches and target-aware gate, push, and rearm behavior
exit state	`main` remains the default target; tagged notes can build and gate on a long-lived integration branch; distillation is bounded and usage-accounted; stall escalation and prefixed live-viz access are complete

Wave 3 made Yesod a recoverable multi-project factory. Wave 4 addressed the next scaling pressure: not every coherent unit of work should merge directly to `main`.

The immediate consumer was the AWS-hosted Yesod roadmap, whose features need to accumulate across several notes before they are ready to graduate. The factory needed a way to preserve the full arm, claim, gate, and merge discipline while directing those changes to an integration branch. At the same time, LLM distillation and factory monitoring needed boundaries of their own: bounded prompts, measured provider usage, and a human notification when the factory truly stalls.

The wave's theme is therefore controlled separation. Work can separate from `main` without escaping the refinery, and intelligence can remain useful without becoming an unbounded background cost.

E.4.2. A Note Can Choose Its Integration Branch

The main feature uses the existing JSONB note-tag system. An untagged note behaves exactly as before. A note tagged `branch:<name>` opts into a separate integration target:

1. The runner resolves the tag and bases the worktree on `origin/<name>`. If the branch does not exist, it creates it from the current `origin/main` and pushes the new branch.
2. When the note enters the merge queue, the resolved target is copied into a new `merge_queue.target_branch` column. `NULL` means `main`.
3. The worker merges the note branch into that snapshotted target, runs the gate against the resulting tree, and pushes the target branch on green.
4. A tagged note never pushes its result to `main`. Graduating the integration branch to `main` remains a deliberate human action.

This snapshot is the critical safety boundary. The note's tags can change while work is queued; the gate destination cannot. The worker gates the branch that the queue row committed to, rather than re-reading mutable policy at the most consequential moment.

The change spans the full lifecycle rather than adding a routing hint at one layer:

Lifecycle point	Change	Evidence
tag resolution	validate one `branch:<name>` target and reject conflicting branch tags	`615305c7`, `note_tags.py`
worktree creation	create or fetch the integration branch and use it as the base	`615305c7`
queue admission	persist `target_branch` at enqueue time with migration `0038`	`23636825`
gate and push	merge, verify, gate, and push to the snapshotted target	`bc02b5b1`
repair	rearm a tagged note against its target instead of assuming `main`	`e714b843`
compatibility	tolerate old claim rows that have no tag data	`0cff2f10`

The default path stays boring on purpose. Existing untagged work still targets `main`, the single merge lock remains in place, and branch-targeted merging is not parallelized in v1. The system does not automatically merge an integration branch back to `main`, and it does not yet reconcile `main` forward into every active integration branch. Those omissions are explicit safety boundaries, not missing documentation.

The result is a new unit of factory composition. A roadmap can be decomposed into independently tracked notes, each note can receive normal verification, and the epic can remain isolated until a human decides that its accumulated state is ready for release.

E.4.3. `main` Stayed Releasable

Branch targeting changes the meaning of "merged." A note can be fully gated and integrated without changing the product branch. That is valuable for large epics, but only if the queue retains an immutable account of where the work went.

The new queue column provides that account in the same durable record that already carries the note branch, gate state, and retry history. The worker's merge and push paths now report the target explicitly, and refinery rearm checks merge cleanliness against that target. An operator inspecting a queue row can answer "which branch is this work trying to change?" without replaying tag resolution or reading a prompt.

This is the branch-level counterpart to Wave 3's crash-only leases. Wave 3 made ownership recoverable after a process dies. Wave 4 makes destination ownership recoverable after policy changes. In both cases, the system takes a decision at a safe boundary and persists it before doing irreversible work.

E.4.4. Distillation Became a Bounded Maintenance Job

The factory's LLM summaries had grown from a convenient lookup into a catalog- wide maintenance surface. The distillation work put an explicit budget around that surface:

- select current usage notes, open work, and recent completed work rather than sending the entire historical note log;
- cap the context and model output sizes;
- include the model and exact context in cache identity, so a summary is not reused after its inputs or model change;
- record provider input/output usage in PostgreSQL through migration `0032`;
- render cached summaries through the maintenance command and run the broad workflow nightly or manually, rather than triggering an LLM call on every push.

This makes "current understanding of the catalog" an observable, budgeted product feature. It also improves the quality of the summary: superseded and terminal history remain available to the database, but stale history does not crowd out the live work in every prompt (`c8308820`, `ys-yes-a2xn`).

The rollout itself supplied a useful warning. The first refinery attempt for the distillation branch encountered a migration-number conflict and unrelated baseline failures, producing a gate red even though the change was not yet on `main`. The branch was repaired, rechecked against the current base, and its retry budget reset before integration. A bounded maintenance job still needs the same merge discipline as product code.

E.4.5. A Stall Now Pages a Human

The propulsion daemon previously nudged agents when it found idle or stalled work. Wave 4 adds a separate human escalation: when the factory meets its stall criteria, eSPS invokes the `email-stephen-devfactory` process through `fmail` (`a41930cc`, `ys-yes-apq1`). This is intentionally not another agent nudge. Agent mail coordinates work; the human-facing channel reports that the factory itself needs attention.

The wave also narrows the runner's claim fast path to the intersection of armed and ready Beads (`48face33`). That avoids ranking work that cannot be claimed and makes the scheduler's hot path reflect actual eligibility rather than catalog volume.

Dispatch received a small but explicit capability update as well: the `codex-gpt-5.6-sol-max` lane exposes Sol at maximum reasoning effort (`e4ad6d1e`). Together these changes make resource choice and escalation visible policies instead of accidental behavior hidden in a daemon loop.

E.4.6. The Proxied Dashboard Became Complete

Wave 3 made the dashboard's links and WebSocket path reverse-proxy-aware. Wave 4 fixed the remaining custom API choke point. `refineries.html` was still calling six `/api/...` endpoints through an absolute-path helper, so the page worked directly but showed an endless "connecting" state under `/yesod/viz/refineries.html`. The fix derives the API base from

the current pathname inside the page-level helper, with tests covering the prefixed and direct forms (`06a6e5c8`, `ys-yes-iq28`).

This is a small patch with a large operational consequence. A dashboard that loads HTML but cannot reach its data source is more misleading than a page that is visibly unavailable. URL construction now belongs to the layer that knows how the URL is being assembled, while the server-side prefix rewriter handles ordinary HTML and redirect surfaces.

The final watchdog test fix belongs to the same "red means broken" standard. The supervisor overrun test had crashed an xdist worker during an unrelated front-end gate. The test now joins its watchdog thread inside the `os._exit` patch, keeping the test's process-control seam contained (`deda1f07`, `ys-yes-2ch8`). A front-end change should not be rejected because a timing- sensitive test can kill its worker.

E.4.7. What Shipped, and What Stayed Out

Area	Shipped result	Boundary at wave exit
integration work	tagged notes, auto-created branch bases, queue target snapshots, target-aware gate/push/rearm	branch graduation to `main` is human-controlled
safety	immutable merge destination and legacy-row compatibility	no parallel branch merging or automatic main-to-epic synchronization yet
LLM maintenance	status-aware bounded context, output caps, model-aware cache, usage metrics, scheduled rendering	distillation still depends on the provider and its recorded usage
propulsion	runner claim fast-path narrowing and human stall email	paging is escalation, not automatic diagnosis or repair
dashboard	prefixed Refineries API requests work behind the deployed path	direct and proxied surfaces still need parity tests as new pages appear
dispatch	Sol max-effort lane and explicit lane cleanup	lane economics still require observation in real work

E.4.8. The New Exit Criterion

Wave 3 asked whether the factory could recover from process, host, repository, credential, and network failure. Wave 4 asks whether it can grow a coherent feature without forcing every intermediate state onto `main`, while keeping its own intelligence and monitoring within explicit budgets.

The next meaningful proof is an end-to-end integration-branch epic: several tagged notes should build on one another, each should gate against the snapshotted target, and the final branch should graduate to `main` only through an explicit human review. In parallel, distillation usage and stall paging should be observed over a maintenance cycle rather than judged only by unit tests.

The durable lesson is that a factory needs more than a merge queue. It needs a safe place for work that is not ready for release, a durable record of which place was chosen, and a bounded amount of intelligence watching the whole system. Integration branches provide the first capability; budgets and human escalation keep it governable.

E.5. Wave 5: Dogfooding the Factory on TTRAC

E.5.1. Wave Summary

Field	Value
purpose	use a real cross-language repository as a compact end-to-end demonstration of planning, dispatch, dependency-aware execution, gating, and merge
integration window	July 27, 2026; the completion checkpoint was recorded at `2026-07-28T00:55Z`
scope	one macOS tray foundation, seven demo feature requests, and three adjacent CLI/API notes
result	the factory reported all 11 TTRAC notes done and merged (`ys-fac-1wha`, `ys-fac-v7bg`)
principal proof	Yesod coordinated Python API work with Swift client work, including rework and merge ordering, instead of treating a green Python test run as the whole product gate
exit state	the demo slice was merged; a TTRAC-specific composite gate was added, while generalized dependency scheduling and broader Swift-lane support remained follow-up work

Wave 4 was about giving work a safe place to accumulate away from `main`. Wave 5 was smaller and more concrete: run that factory against a product that was itself heterogeneous. TTRAC combines a FastAPI/Postgres service with a macOS Swift tray application. The demo therefore tested the factory at the seam where repository identity, feature dependencies, model routing, platform tooling, and merge discipline all meet.

This was not a new platform subsystem. It was a dogfood slice with a useful stopping condition: build enough of the tray application to demonstrate a real workflow, put every change through the refinery, and record what the exercise reveals about the factory.

E.5.2. The Demo Had a Real Dependency Shape

The foundation note, `ys-ttr-e252`, created the macOS application scaffold: `MenuBarExtra`, the API client and Codable models, the client selector, daily and weekly totals, quick actions, and the initial `TimerManager` state. The remaining seven UI notes then filled in a coherent demo surface:

- idle and display-sleep detection, with timer suppression and wake recovery (`ys-ttr-e1e0`);
- preferences (`ys-ttr-vkiv`) and a report window (`ys-ttr-2q48`);
- five-minute ScreenCaptureKit screenshots with upload and local backup (`ys-ttr-q08k`);
- a six-minute confirmation popup (`ys-ttr-qoxc`);
- quick note entry (`ys-ttr-9ttq`); and
- an invoice wizard (`ys-ttr-rpmv`).

The plans were not merely a list of independent screens. The planner verified the existing API contracts and made the dependency order explicit. The timer core followed `e252 → e1e0 → q08k → qoxc`; the window work shared the foundation but used separate presentation decisions, including client-side PDFKit for the report and API-side ReportLab for invoices (`ys-fac-jcnf`, `ys-fac-a1ik`).

That shape mattered to the factory. A screenshot worker cannot safely extend a `TimerManager` that another worker is changing without either a dependency boundary or a rework path. The demo made that interaction visible instead of leaving it as an assumption in a plan.

E.5.3. Dispatch Became Observable Product Behavior

The dispatcher armed all seven feature notes after planning, with model lanes chosen by expected difficulty: easy work went to Kimi, medium work to DeepSeek, and advanced work to Claude Opus (`ys-fac-24yh`). The foundation was handled first because the other workers needed its Swift project, API client, and timer seam.

The run also showed that "parallel" does not mean "uncoordinated." The factory handled three rework cycles, including a `ttracApp.swift` conflict in the quick note path and a `TimerManager.swift` conflict in screenshot work (`ys-fac-vskg`). Those were not invisible model retries. They became part of the operational record and were resolved through the same refinery path as the initial work.

The final completion record included the adjacent support changes that made the demo easier to operate: `ttrac --version`, `ttrac-migrate --version`, and a retroactive screenshot-to-time-entry update endpoint. The result was reported as 11 of 11 TTRAC notes merged, rather than merely seven agents having been started (`ys-fac-1wha`, `ys-fac-v7bg`).

E.5.4. The Demo Found a Gate That Was Too Narrow

The most valuable discovery was a failure in the definition of green. TTRAC's refinery profile ran pytest, but TTRAC also contained a Swift package. A Swift 6 compile error could therefore pass the Python gate and enter `main`. The issue was captured as `ys-yes-6lfp` / bead `yes-7hgw`, and the profile was changed to a composite command:

```
uv run pytest -q -p no:cacheprovider && swift build --package-path mac
```

The implementation landed in the default profile and in an idempotent profile migration (`805b3fd4`, `824b602f`, `e5b4f4cd`), with regression coverage proving that both halves of the gate are present (`69312146`). The repository path was seeded with the profile as well, so a fresh database and an existing factory agree about which TTRAC checkout they are gating.

This is the central lesson of the wave. A repository is not defined by the language of its largest directory. The gate has to describe the product's buildable surface. TTRAC made that rule unavoidable because the Python half and the Swift half were both part of the feature being demonstrated.

E.5.5. Operations Were Part of the Proof

The demo also exercised the less glamorous parts of the factory:

- a launchd environment override forced the wrong gate backend and had to be corrected before the local flow could proceed;
- a slow gate held the dependent TTRAC chain, making the ordering constraint visible to the dispatcher;
- orphaned and stale queue work required re-arming and re-gating rather than assuming that an earlier worker's state was still authoritative; and
- parallel Swift edits produced real merge conflicts, which the dispatcher tracked as rework rather than silently discarding.

The factory notes record both the friction and the recovery. That is more useful than a clean demo transcript: the system was judged on whether it could explain why work waited, repair the queue state, and eventually produce a merged result. The follow-up dependency note, `ys-yes-daf7`, captures the generalization still needed: enforce one dependency layer per concern and make readiness efficient enough that a small demo does not require manual queue archaeology.

E.5.6. What Shipped, and What Stayed Out

Area	Shipped result	Boundary at wave exit
product slice	tray foundation, idle suppression, preferences, report, screenshots, popup, quick notes, and invoice flow	this was a demonstration surface, not a claim of production-ready macOS distribution
factory use	seven dependent UI notes planned, routed, reworked, gated, and merged	dependency ordering was partly planner- and dispatcher-mediated; a general dependency scheduler remained future work
validation	TTRAC's profile chained Python tests with the Swift build	the composite gate was TTRAC-specific; other mixed-language repositories still need explicit profiles
operations	stale backend configuration, queue delay, orphan work, and merge conflicts were recovered in the run	recovery was demonstrated, not yet reduced to a single automatic repair policy
completion evidence	11/11 TTRAC notes reported done and merged	no deployment or end-user rollout was implied by source integration

E.5.7. The New Exit Criterion

Wave 4 asked whether an epic could accumulate safely without forcing every intermediate state onto `main`. Wave 5 asked whether the factory could carry a small, real, cross-language product slice from plan to merged result.

The answer was yes, with an important qualification: the factory's correctness boundary is only as complete as the repository profile behind it. The demo finished not when the agents had produced plausible Swift files, but when the Python tests, Swift build, dependency order, conflict repairs, and queue state all agreed that the result was mergeable.

The durable lesson is that dogfooding is a form of systems testing. A compact demo can expose a missing gate dimension, an implicit dependency, or a queue repair gap faster than a broad roadmap can. TTRAC gave Yesod a product small enough to finish in one wave and heterogeneous enough to make the factory's assumptions visible.

E.6. Wave 6: Identity, Idleness, and the Quiet Factory

E.6.1. Wave Summary

Field	Value
purpose	make “nothing is happening” a safe, observable factory state; stop autonomous token use after sustained idle; make server-backed Beads identity and note linkage fail closed instead of disappearing
development window	opened August 2, 2026; this entry records the opening checkpoint rather than a completed exit
baseline	`b8ca1294` on `main`
identity hardening branch	`beads-identity-ys-yes-zdry`; 5 commits through `9708f916`
branch change surface	19 files, 1,647 insertions, 56 deletions
triggering incidents	an orphaned ALS feature note exposed two clones with a stale Beads guard token; a manual shutdown request exposed the absence of a deterministic idle-to-off transition
current operating state	ALS and ALS-refinery are usable with 34 Beads issues each; eSPS is off; the hardening branch is not yet on `main`; automatic idle shutdown and message expiry remain open
exit condition	identity safeguards merged and audited across the workspace fleet; sustained authoritative idle stops every autonomous service exactly once; factory mail is purged; an explicit restart begins with an empty message session

Wave 5 proved that Yesod could carry a real cross-language product slice to a merged result. Wave 6 begins with a less glamorous question: **what should the factory do when there is no result left to produce?**

An autonomous system can waste resources while appearing quiet. A propulsion loop can keep prompting agents, a dispatcher can keep reconsidering an empty or blocked queue, and old coordination messages can be mistaken for current intent after a restart. At the same time, machine-local state can look like ordinary source code and spread a bad identity to every clone.

The two failures share a boundary problem. Durable truth and disposable execution state were not separated sharply enough. Wave 6 makes that separation explicit: notes, Beads issues, commits, and audit events are durable; agent mail, sessions, leases, and clone-local guard tokens are not.

E.6.2. Idle Is a Derived State, Not an Empty Screen

On August 2 the operator disabled eSPS to stop further autonomous prompts. The service status showed `off` with no daemon PID. That was the correct immediate containment action, but it was not yet a factory-level idle protocol. Turning off propulsion alone does not prove that a runner, gate, merge, or local agent is not still working.

The open feature request `ys-yes-6qs5 / yes-4pgc` defines the stronger contract. An idle decision must be derived from authoritative state across:

- live factory agents;
- claimed and in-flight dispatch runs;
- refinery gates and merges;
- actionable queued work; and
- a configurable grace period that spans normal gaps between steps.

Only sustained idle may trigger shutdown. The shutdown itself must be idempotent and must disable eSPS, factory roles, the mail broker, local dispatcher, refinery supervisor, and seat keeper. Active work blocks the transition. A status and dry-run surface must show the exact evidence behind the decision, and the audit record must be produced without spending another LLM call.

This is a cost-control feature, but it is also a correctness feature. "No work is visible to me" is not evidence that no work exists. The factory may go dark only after all of its ownership records agree.

E.6.3. Coordination Has a Lifetime

At the opening checkpoint, the catalog still held 1,914 factory mail rows. Those rows were useful coordination artifacts when they were written; they are not a durable knowledge base. Preserving them indefinitely makes a later session vulnerable to stale nudges, obsolete hand-offs, and accidental revival of work that has already been resolved elsewhere.

Wave 6 therefore assigns each information surface a lifetime:

Surface	Lifetime	Reason
Yesod notes and topics	durable	operator intent, decisions, incidents, and reusable knowledge
Beads issues and dependencies	durable	executable work identity and dependency state
Git commits and gate evidence	durable	implementation and verification record
audit event for an idle shutdown	durable	explains why and when autonomous execution stopped

Surface	Lifetime	Reason
agent mail and message rows	bounded and purgeable	transient coordination, not a source of truth
agent sessions and leases	process-scoped	execution ownership that must not survive its owner
`.beads/metadata.json`	clone-local	guard token binding one checkout to one server database

Idle shutdown is the natural garbage-collection boundary. Once the factory has proven that no active work remains, it can wipe transient messages and stop the processes capable of consuming them. A later explicit start should recreate a clean session rather than replay the previous factory's conversational debris.

E.6.4. The ALS Incident Exposed an Identity Boundary

The immediate Beads incident began when `yesod tool als feature-request` created note `ys-als-qh64` but could not create its linked issue. The note was left without a Beads identity and was therefore invisible to the dispatcher. Diagnosis showed that both the main ALS clone and its refinery clone failed all `bd` operations with a project identity mismatch.

The server database was healthy and contained 33 issues. Its `metadata._project_id` was `705f983c-c8f7-4ba6-b975-60772bfd6c3f`. Both clones instead carried the local token `2046f420-abb4-4503-a5e1-408413288ac4`, an identity minted by a later `bd init`. The guard correctly refused the connection: accepting either side silently would have risked writing one project's issues into another project's database.

The disease was propagation, not the guard. `.beads/metadata.json` had been tracked in Git, untracked once, and then swept back into an unrelated feature commit after its ignore rule disappeared. A machine-local token became repository history and spread through ordinary pulls to every clone.

Recovery followed the authority boundary in the safe direction:

1. query the server metadata and issue count first;
2. establish that the populated server database is the source of truth;
3. replace only the clone-local guard tokens;
4. untrack and durably ignore the metadata file; and
5. retry the missing note-to-Bead link.

The orphaned note became issue `als-qwf`, raising the database count to 34. Both ALS workspaces then returned to `ok`. No issue history was copied or reinitialized.

E.6.5. The Server Became Canonical

Bug report `ys-yes-zdry` / `yes-vr8b` produced a five-commit hardening branch. Its central rule is simple: for a server-mode workspace, the canonical project identity lives in the server database. A runner or provisioning path may seed an identity only when the database is genuinely empty. If a populated database has no `_project_id`, Yesod stops rather than inventing one.

The branch carries the rule through the whole lifecycle:

Boundary	Safeguard
provisioning	read the existing server identity; generate only for an empty database
runner worktree staging	write the server's identity into local metadata instead of calling `uuid4()`
diagnosis	`yesod beads doctor` compares local and server IDs and reports tracked or unignored metadata
repair	`yesod beads repair-identity TOOL --from-server` previews by default and writes only with `--apply`
source control	root ignore rule, pre-commit rejection, CI rejection, and worker instructions keep metadata out of commits
missing linkage	a PostgreSQL outbox leases and retries failed note-to-Bead creation with bounded backoff

The last safeguard closes the exact invisibility gap exposed by ALS. Note creation and outbox enrollment occur in one catalog transaction. A temporary Beads failure no longer turns an actionable request into an untracked orphan; runners retry the recorded failure, and successful linkage removes the outbox row.

The branch contains regression coverage for canonical identity resolution, safe repair, Git guards, runner staging, and outbox retry behavior. At this checkpoint it is pushed through `9708f916`, but it is not yet part of `main`. That distinction matters: implemented on a branch is evidence of progress, not fleet protection.

E.6.6. One Populated Database Still Needs a Deliberate Migration

The SJBGTD workspace demonstrates why the new code fails closed. Its server database contains 151 issues but has no `_project_id`. There is no safe value for an automated repair command to guess. The local token may be correct, but it must first be proven against the long-lived workspace and every healthy clone.

The handoff plan requires quiescing writers, capturing server and clone evidence, taking a recoverable Dolt snapshot, establishing one canonical UUID, inserting it server-side through

an authorized path, and only then repairing each clone from the server. Backup configuration is a separate least-privilege operation; broad remote-admin authority must not be granted to `yesoduser` merely to make `bd dolt push` succeed.

This is intentional friction. A repair tool is safe because it refuses the case in which the operator has not yet established the truth.

E.6.7. What Has Landed, and What Remains

Area	Evidence at this checkpoint	Required before wave exit
ALS recovery	both clones use the server identity; the orphan note is linked as `als-qwf`; each reports 34 issues	retain healthy identity after normal clone and refinery activity
identity implementation	five commits on `beads-identity-ys-yes-zdry` through `9708f916`	merge, deploy, and run the fleet audit under the released Yesod
Git boundary	ALS metadata is untracked and ignored; branch adds hook, CI, and worker guards	audit every registered repo and refinery clone for tracked metadata
linkage durability	branch adds a leased outbox and runner retry path	demonstrate recovery from a real transient failure after deployment
immediate cost containment	eSPS is off with no daemon PID	implement and prove whole-factory idle detection and idempotent shutdown
message lifecycle	ephemeral-mail contract is recorded in `ys-yes-6qs5`	purge existing transient rows, enforce retention, and prove clean restart
SJBGTD	151 server issues remain reachable; migration plan is documented	establish `_project_id`, align all clones, and complete a recoverable backup sync

E.6.8. The New Exit Criterion

Earlier waves made the factory capable of continuing after failures. Wave 6 adds the complementary capability: stopping cleanly when continuation has no value.

The wave is complete only when two proofs exist. First, every server-backed workspace must derive identity from a known database of record, while every clone-local guard stays outside Git and every failed note linkage remains recoverable. Second, an idle factory must be able to explain that it is idle, wait through a bounded grace period, stop every autonomous component exactly once, remove transient coordination messages, and remain quiet until a human explicitly starts a clean session.

The durable lesson is that autonomy needs an off-state as carefully designed as its run-state. A factory that cannot distinguish durable truth from transient coordination will either forget important work or preserve activity forever. The quiet factory does neither.

E.7. Wave 7: The Legible Factory

E.7.1. Wave Summary

Field	Value
purpose	make the factory understandable as a chronological, filterable story; distinguish work that is rejected from work that is accepted but intentionally parked; preserve grounded plans without accidentally scheduling them
development window	opened August 7, 2026; this entry records the opening checkpoint rather than a completed exit
baseline	`162de6dd` on `main`
triggering friction	an AWS-hosted roadmap produced executable plans that were not ready to run; Yesod had no `deferred` lifecycle state; the existing `/viz/live-events.html` page exposed a narrow event rail rather than a human-scale activity stream
shipped foundation	catalog activity history, note-status history, a standalone live-events page, cursor-based event backfill, and the `yesod factory status` rollup
open work	`ys-yes-b3iy` adds the `deferred` lifecycle state; `ys-yes-xqhi` turns the live-events prototype into a leveled, multi-tool operator stream
exit condition	from one page, an operator can tell what happened, at the desired level of detail, for selected tools; accepted parked work is represented as `deferred`, remains non-dispatchable, and can return to `open` with its plan and history intact

Wave 6 asked whether the factory could become quiet without losing durable truth. Wave 7 asks the complementary question: **when the factory is active, can a human under-**

stand what it is doing without reconstructing the answer from its databases and dashboards?

Yesod already records more state than most software factories. Notes have lifecycle histories. Beads record executable structure and dependencies. Dispatch runs, leases, gates, merge jobs, agents, and mail each expose their own transitions. The problem is not missing state in the abstract. The problem is that state is distributed across surfaces with different vocabularies, retention rules, and levels of detail.

A human does not experience those surfaces as separate subsystems. They ask what just happened, what is running, what is waiting, what was deliberately put aside, and whether a transition matters. Wave 7 treats those questions as a product contract. The factory becomes legible when its durable records form a coherent story at the level of human attention.

This opening checkpoint does not declare Wave 6 complete. Identity hardening, idle shutdown, and transient-message cleanup retain their own exit obligations. Wave 7 begins from the state they helped clarify: durable truth and disposable execution are different things, and an operator surface must say which kind of evidence it is showing.

E.7.2. The Factory Had State but No Story

The command `yesod factory status`, tracked by `ys-yes-q0e7`, consolidated an important set of operational facts. It made queue, supervisor, runner, gate, and fleet state available through one rollup rather than a sequence of private queries. That is a major improvement in observability, but a status snapshot answers only one class of question: what appears to be true now?

An operator also needs causality. Why did a note leave `open`? Which Beads were created when it was grounded? Did a worker claim the task, or did a dependency change make it eligible? Is a gate merely slow, actively running, retrying an infrastructure failure, or finished red? A snapshot can show the latest state while concealing the transitions that explain it.

Yesod's current interfaces make that explanation possible but expensive. The catalog landing page has an activity stream for catalog mutations. Note detail can show lifecycle history. The live-viz event log carries Bead and dependency events. Refineries and run pages expose gate and execution timelines. An operator can assemble the story, but only by knowing which surface owns each piece and mentally joining their identifiers.

Legibility is the removal of that mental join. It does not mean flattening all events into an undifferentiated log. It means giving each event a stable identity, time, subject, actor, tool, kind, and human-readable meaning, then allowing the operator to choose how much machinery to see.

E.7.3. Parked Work Is Not Rejected Work

The AWS-hosted Yesod roadmap exposed a second vocabulary gap. Phase 1 was decomposed into four fresh feature records:

- external PostgreSQL portability (`ys-yes-9xio`, root Bead `yes-fq93`);

- authenticated HTTP mutation boundaries (`ys-yes-v4cl`, `yes-ozj5`);
- provider-isolated MicroVM runtime flavors (`ys-yes-zkae`, `yes-0isq`); and
- a clean-install acceptance path (`ys-yes-81wu`, `yes-pip6`).

Each root has three file-scoped child Beads, an internal dependency chain, acceptance criteria, and a targeted gate. The first three features can proceed independently; the clean-install proof depends on all three. This is executable planning, not a list of aspirations.

It is also work that should not run yet. The roadmap is exploratory, the AWS delivery decision has not been made, and placing twelve well-specified child tasks in front of an autonomous dispatcher would confuse preparedness with authorization.

Yesod's lifecycle vocabulary had no honest state for that combination. `open` means actionable work available for planning or dispatch. `wontfix` means the work was rejected or abandoned. A dependency-derived block means the work would proceed if its prerequisites completed. A hold or missing arm is an execution-control fact. None means "we accept this direction, have preserved a good plan, and intentionally do not want it scheduled."

Until a better state exists, the four AWS records are temporarily `wontfix`, with comments saying they should become `deferred` once `ys-yes-b3iy` ships. Their root Beads are closed, while the child trees, tags, dependencies, and plans remain preserved. The workaround is safe for dispatch, but semantically wrong. A future reader counting rejected work cannot distinguish an abandoned idea from a deliberately parked roadmap phase without reading the comments.

The proposed `deferred` state repairs that meaning. It represents accepted work that is intentionally outside the active backlog. It must never be claimable, dispatchable, or automatically armed. Returning it to `open` must preserve its tags, Bead linkage, dependency graph, metadata, plan, and history. `wontfix` then regains a precise meaning: no current intention to implement.

E.7.4. Grounding Became a Durable Asset

The temporary AWS state reveals a broader design principle: planning and scheduling are separate authorities.

A grounded feature is valuable before a worker is allowed to touch it. The note records the product boundary and acceptance test. Its root Bead gives the feature one merge identity. Three to ten child Beads give an agent concrete, file-level steps. Bead dependencies order work inside that merge boundary; note dependencies order independently mergeable features. Together they make the plan inspectable, reusable, and cheap to revalidate later.

None of that implies permission to spend tokens, launch a MicroVM, write a branch, or alter AWS. Grounding answers "what would we build?" Dispatch answers "should the factory build it now?" Conflating them makes careful roadmap work dangerous: the better the plan, the more likely an autonomous system is to mistake it for an immediate command.

`deferred` turns a grounded plan into a durable portfolio asset. The plan can survive a change in priorities, a missing provider capability, or a budget decision without polluting the active queue or being mislabeled as rejected. When it is revived, a planner does not start over. The

planner revalidates the baseline, file paths, dependencies, and acceptance gate, then returns the note to `open`.

E.7.5. From Event Rail to Activity Stream

The existing page at `/viz/live-events.html` is the natural surface for the other half of this wave. Its source, `yesod/live-viz/live-events.html`, is a small React page backed by `All-BeadsAdapter`. It loads snapshot events, opens a live stream, and renders a newest-first rail. The page recognizes generic audit events, Bead events, and dependency events. It shows a connection indicator, relative and wall-clock time, actor, target, and a short message.

That is useful infrastructure, but it is still a prototype rather than a factory activity product. It has no multi-tool selector, no event-kind filter, no expandable structured details, and no distinction between a milestone and a heartbeat. It keeps a client-side window of 500 events. Its pause control stops the relative-time clock but not event ingestion. Its typed-event handler currently risks appending the same event twice. A demo-data toggle remains in the production page.

The feature request `ys-yes-xqhi` does not call for another page. It names this page as the implementation target and asks the existing catalog, status, live-viz, Bead, refinery, and gate sources to converge on a normalized event envelope. Initial history and live delivery must use the same identity and filter semantics. Cursor recovery must make retention gaps visible rather than silently presenting an incomplete story.

The product distinction is selectable detail:

Level	Operator meaning	Representative events
high	meaningful milestones	tool onboarded; note created; every note lifecycle change; dispatch, gate, merge, deployment, or major failure
medium	work structure and decisions	root or child Bead created, claimed, or closed; dependency changed; planning completed; runner assigned; gate batch formed
low	diagnostic mechanics	heartbeat, retry, lease, queue transition, reconciler pass, worker phase, or per-shard gate progress

The levels are cumulative. High shows only milestones. Medium adds the work structure that explains them. Low adds the machinery needed for diagnosis. This is not merely a severity filter: a successful heartbeat is low-detail but not low-importance when diagnosing liveness,

while a routine note transition is high-level because it changes the user's understanding of the work.

The second control is scope. An All Notes-style multi-select tool filter should let the operator watch one repository, a related group, or the whole factory. The same filter must constrain loaded history and new streamed events. A newest-first catch-up view should coexist with a live-follow mode that does not steal scroll position while the operator is inspecting older evidence.

E.7.6. Attention Became a Control Plane

Raw telemetry optimizes for completeness. Human attention does not. It is bounded, changes with the task, and is easily defeated by an event source that treats every database write as equally prominent.

The high, medium, and low contract is therefore a control-plane decision. It defines which transitions deserve immediate human awareness and which remain available when the operator asks for more resolution. The classification must be canonical per event kind rather than guessed independently by each page. Otherwise a gate retry will be a milestone in one view, a diagnostic in another, and absent from a third.

The activity stream also needs a strict disclosure boundary. Low-level does not mean raw. Event details may contain stable identifiers, structured error classes, durations, counters, and links to bounded logs. They must not contain provider secrets, reusable credentials, raw prompts, or unbounded stdout. The stream is an index into evidence, not a second log archive.

This makes the activity page more than a visualization. It becomes the place where durable intent and disposable execution meet without being confused. A `deferred` transition is a high-level event because it changes portfolio intent. A MicroVM heartbeat is low-level because it describes one disposable runtime. Both remain auditable, but they compete for attention differently.

E.7.7. What Has Landed, and What Remains

Area	Evidence at this checkpoint	Required before wave exit
current-state rollup	`yesod factory status` consolidates major operational surfaces (`ys-yes-q0e7`)	link status facts to the events that produced them
catalog activity	the `yesod serve` landing page records catalog creation and mutation activity (`ys-yes-zsy3`)	normalize catalog events with factory events instead of maintaining a separate story

Area	Evidence at this checkpoint	Required before wave exit
live page	`/viz/live-events.html` exists and consumes snapshot plus streamed Bead events (`ys-yes-u150`)	replace prototype controls and narrow normalization with the leveled, filterable operator experience
reconnect safety	cursor/backfill and snapshot recovery infrastructure exists (`ys-yes-ih57`)	prove initial history plus reconnect has no silent gaps or duplicate rendering across the unified envelope
parked AWS work	four Phase 1 features have grounded three-child trees and explicit cross-feature ordering	represent them as `deferred` rather than temporary `wontfix`, then revalidate rather than replan when revived
lifecycle vocabulary	`wontfix`, holds, dependencies, and planning each have distinct existing meanings	implement `deferred` across constraints, writers, filters, views, exports, search, distillation, and dispatch exclusion (`ys-yes-b3iy`)

E.7.8. The New Exit Criterion

Earlier waves proved that the factory could carry work, survive failure, merge through a remote gate, and stop when no action remained. Wave 7 is complete when the operator no longer needs subsystem archaeology to understand those capabilities in motion.

From one page, the operator must be able to answer:

- What happened, in chronological order?
- Which changes matter at a high, medium, or diagnostic level?
- Which tools and work objects were involved?
- What is running, waiting, completed, or intentionally parked?
- Who or what caused each transition?
- Did the stream reconnect cleanly, or is some history unavailable?

At the same time, the lifecycle must tell the truth about intent. Accepted but parked work must be `deferred`, excluded from every execution path, and recoverable without losing its grounded plan. Rejected work must remain `wontfix`. Active work must not become eligible merely because it is detailed.

The durable lesson is that observability is not the accumulation of status pages. A factory becomes legible when its state changes form a coherent, filterable narrative and when its lifecycle vocabulary distinguishes what the system can execute from what the human has authorized it to pursue.

E.8. Wave 8: The Conductor and the Deterministic Refinery

E.8.1. Wave Summary

Field	Value
purpose	replace the split-brain, host-coupled autonomous merge engine with a durable deterministic refinery; move standing control-plane infrastructure off the operator workstation and onto a dedicated Conductor VM
development window	August 7–9, 2026—from the redesign document through Conductor migration, three-lane proof, idempotent repository onboarding, and the self-explaining live refinery surface
triggering incidents	the “5385” page showed a completed green gate as a one-hour zero-pass run; sharded-suite state pollution mislabeled innocent branches; launchd `KeepAlive` made process-level pauses unsafe; control-plane services were still anchored to `pompom`
source baseline	`40a9622c` before the redesign branch; activation began at `94e80a7b`, the Conductor migration merged at `d434f0aa`, host portability reached `86b3dc46`, and the final reviewed/deployed release is `4f5a183b`
core redesign	`refinery-reliability-ys-yes-x802`, 25 commits through `3db9126b`; merged by the normal eight-shard gate at `77d114f5`
redesign change surface	69 files, 28,790 insertions, 1,959 deletions; 24 implementation Beads closed under `yes-0f10` / `ys-yes-x802`
activation hardening	secure capability token, PID-reuse fencing, observable provider cancellation, serial-in-shard MicroVM execution, provider-start evidence repair, and schema-init deadlock fixes through the `94e80a7b` candidate
Conductor migration	`refinery-conductor-qii1`, 17 commits through `6597a7f5`; merged by the deterministic refinery at `d434f0aa`; tracked by `yes-1358` / `ys-yes-qii1`

Field	Value
live topology	`pompom` is an operator client; `seykhl` is only the Proxmox hypervisor; VM 112 `yesod-dispatch` runs the standing control plane; `dertog` serves visualization/read surfaces; `homestar` stores immutable artifacts; AWS MicroVMs run the Yesod gate
production scope	three configured lanes: `yesod` through eight MicroVM shards with one pytest worker per shard; `scr-wrap` through one local serial gate; and `yesod.work` through the exact local `npm ci`, `npm run check`, `npm run build` profile; other projects remain explicitly outside active intake until reviewed
evidence	277 targeted refinery acceptance tests; independent reviewer rechecks; repeated eight-MicroVM gates with zero failures or infrastructure errors; 53 focused fleet/API/browser/migration tests plus an eight-test tripwire follow-up; 59 focused onboarding/source/runtime tests plus two installed-CLI no-op replays; controller and supporting units active with zero restarts; public dashboard HTTP 200 and visually checked; Blob mount verified byte-for-byte; all nine `yesod.work` onboarding checks green; and sentinel `ys-ysw-6kot` reaching `main`, deleting its queue row, and closing `ysw-p9l` automatically
checkpoint type	final operational and source-code exit for the deterministic-refinery/Conductor migration; the factory is active on three reviewed lanes with the legacy supervisor absent
exit status	achieved: the migration is promoted, heartbeat/artifact/fleet visibility are live, the operator inbox is empty, host placement is explicit, a real third-lane terminal write-back completed, and new repositories now have one idempotent command plus a nine-invariant readiness check

Wave 7 asked whether the factory could tell a coherent story. Wave 8 began when the refinery told a story that was coherent, current-looking, and false.

The Refineries page showed a MicroVM whose identifier ended in 5385 as roughly one hour old with zero passing tests. The actual remote job had already finished green: 5,845 passed, 26 skipped, zero failed, in 243.3 seconds. The page had joined fresh VM uptime, a stale pending gate row, and an idle merge pipeline as though they described one live execution.

That incident was not a display typo. It exposed a refinery whose lifecycle authority was distributed across queue labels, process memory, worker rows, merge locks, gate files, provider jobs, fleet snapshots, and several APIs. An LLM supervisor could react to symptoms, but it could not supply fencing, exactly-once effects, or a single answer to "what attempt owns this gate?"

Wave 8 replaced that arrangement and then moved the replacement onto the host where standing infrastructure belongs.

E.8.2. The Run Began With a Real Pause

The first operational decision was to stop the old refinery without damaging the work already in flight. The supervisor was unloaded from launchd, not merely killed or kickstarted. That distinction mattered because `KeepAlive` can relaunch a killed process, while an unloaded job is absent from the service manager and cannot fight the operator's pause.

The pause was taken after the active gate completed. The merge lock was free, no runner was building, and the one queue row labeled `gating` was proven to be stale state with no live gate behind it. This established the boundary for the redesign: no mid-merge interruption, no orphaned lock, and no new work entering while the source of truth changed.

The flake evidence also narrowed the problem. Several innocent notes had been escalated to rework because different full-suite runs failed different test modules, while the same subsets passed locally and under isolated `-n auto` runs. The failure was cross-worker xdist state pollution, not seven unrelated product regressions. The new production Yesod profile therefore preserves parallelism across machines while running one pytest worker inside each MicroVM.

E.8.3. An Attempt Became the Unit of Truth

The redesign's primary object is no longer a queue row or a supervisor cycle. It is one immutable merge attempt in one repository/target lane.

PostgreSQL now holds first-class records for:

- project lanes and their current attempt;
- immutable attempts, candidate SHAs, leases, fencing generations, and terminal outcomes;
- concrete local or MicroVM gate executions;
- append-only attempt events and external-effect evidence;
- bounded artifact metadata pointing into Yesod Blob; and
- an operator inbox for facts the controller cannot safely infer.

The normal lifecycle is monotonic:

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

Retries create successor attempts; they do not reopen terminal records. A worker must present the current owner and fencing token for every mutation. If its lease expires and another controller acquires a higher token, the old process can no longer record progress, terminal evidence, or artifacts.

A lane is repository identity plus target branch. One attempt may push within that lane, while unrelated lanes may prepare and gate concurrently. The global merge loop has become a bounded scheduler over independently serialized lanes.

E.8.4. External Effects Became Reconciled Facts

Git pushes and provider submissions are not database transactions. A process can receive success, lose the response, and die before recording it. Treating that as a normal retry risks a duplicate gate or a repeated push.

The new brokers therefore separate intent, observation, and completion:

1. persist an idempotent external-effect intent;
2. perform the provider or Git operation outside the database transaction;
3. observe the real external state;
4. record confirmed-applied, confirmed-not-applied, or outcome-unknown; and
5. reconcile unknown outcomes by stable identity instead of repeating them blindly.

Local and MicroVM gates implement the same provider contract: submit, find by idempotency key, observe, cancel, and fetch immutable artifacts. A lost submit response can be recovered from the deterministic provider job identifier. A green gate authorizes only the exact candidate SHA it tested; if the target branch moves first, the controller must construct and gate a new candidate.

E.8.5. Review Found Two Missing Invariants

The first implementation passed its focused broker, scheduler, runtime, and UI suite, but review exercised two cases the tests had not actually proved.

First, a provider that still reported `running` one second past the configured deadline left the gate `pending=true` forever. The earlier test covered only a provider job missing at deadline. The fix preserves the final running observation, requests cancellation when supported, observes the cancellation outcome, and then settles the execution under the bounded lost policy. A remote provider cannot extend the controller's reconciliation deadline merely by continuing to say "running."

Second, gate idempotency was global over candidate SHA and profile. Two different project lanes with the same SHA/profile could collide, causing the second lane to reuse or reject the first lane's execution. The identity now includes repository/target lane identity, so equal candidates in independent projects produce independent gates and provider jobs.

Those fixes landed as `c58994e4` and `a91dfa86`; `3db9126b` recorded them in the acceptance matrix. The final targeted acceptance run reported 277 passing tests.

E.8.6. Activation Required More Than Green Tests

The reviewer approved merge but withheld activation until three operational boundaries were hardened.

Boundary	Hardening
active capability	the token must be a regular mode-0600 file, owned by the service identity, with one hard link and exact reviewed contents
local process ownership	the controller records and checks process identity rather than trusting a reusable PID after a crash
overdue provider work	cancellation is an explicit external effect whose observed outcome is retained; "marked lost" is not treated as equivalent to "provider stopped"

The activation branch also changed the Yesod MicroVM profile from nested xdist to eight machines with `-n 1` inside each guest. Parallelism remains eight-wide at the infrastructure level, where each shard has isolated process and database state. The old shared-process pollution channel disappears.

Later sentinel work corrected two more activation findings. Provider binding had written receipt acceptance time into `started_at`, conflicting milliseconds later with the provider's real start time. Schema initialization could also deadlock when legacy intake and a newer process raced through migrations. Both were repaired before the reviewed `94e80a7b` controller candidate was activated.

E.8.7. The Gate Is Remote; the Controller Is Not the Test Host

"Deterministic refinery" describes control semantics, not local execution. The standing controller runs on `yesod-dispatch`, but the Yesod gate runs on eight disposable AWS Lambda MicroVMs. Each guest receives the exact candidate, runs one serial pytest worker against its isolated in-image PostgreSQL, and reports a provider observation. The controller aggregates those observations into one durable gate execution.

This is the performance and reliability compromise the flake evidence called for:

- parallel across eight isolated machines;
- serial inside each machine;
- no workstation CPU contention;
- no cross-worker process state;

- no silent local fallback when the remote provider is unavailable; and
- a durable deadline and cancellation path when a remote job does not settle.

The `scr-wrap` lane remains a one-shard local serial gate because its reviewed profile is small. Local is a provider choice per lane, not a global fallback for Yesod.

E.8.8. The Workstation Stopped Being the Server Room

The initial activation still left a structural contradiction: the new controller and its support jobs were launch agents on `pompom`. The Mac Studio is a client workstation. Login state, GUI sessions, battery/workstation maintenance, and user experiments should not decide whether the software factory can merge.

The Conductor migration, tracked by `ys-yes-qii1` / `yes-1358`, established a dedicated placement model:

Place	Authority and responsibility
`pompom`	interactive operator client: source checkouts, the Yesod CLI, interactive credentials, and preserved-but-disabled rollback plists; no standing Yesod, sjbis-gog, refinery, or OpenCode service
`seykhl`	Proxmox hypervisor only; owns VM 112 configuration and storage, but runs no Yesod application process directly
`yesod-dispatch`	VM 112, the Conductor; owns deterministic refinery control, gate proxying, fleet telemetry, factory dispatch, maintenance timers, status reporting, and human/question bridges
`dertog`	visualization and read surface: Yesod HTTP/API, live-viz WebSocket, public Refineries page, and the human-facing sjbis daemon; it does not claim or merge work
`homestar`	Synology NFS artifact store at `/volume1/yesod-blobs`, mounted by the Conductor at `/mnt/nas/yesod-blobs`
AWS Lambda MicroVMs	disposable gate workers; no durable authority survives an instance

On `pompom`, sixteen `com.yesod.*`, `com.sjbis.gog`, and `ai.opencode.server` launch-agent plists remain available for deliberate rollback, but every label is unloaded and persistently disabled. The stale sjbis bridge PID file was removed only after its PID was proven absent. A login or reboot can no longer resurrect the old refinery through `KeepAlive`.

On seykhl, VM 112 is configured with two cores, 4 GiB RAM, a 20 GiB local-lvm disk, onboot=1, and startup order 20. The application boundary is the guest: /etc/pve/nodes/seykhl/qemu-server/112.conf describes the VM; it is not a Yesod deployment directory.

E.8.9. The Conductor Has an Explicit Filesystem Contract

Inside yesod-dispatch, immutable code and mutable authority are separated:

Path	Purpose
/srv/yesod/releases/<commit>	immutable source plus locked environment for one reviewed commit
/srv/yesod/current	controller/proxy/dispatcher release; final reviewed release 4f5a183b
/srv/yesod/ops-current	independently pinned auxiliary-service release; live at 233e00b8 for this checkpoint
/var/lib/yesod	mutable repositories, controller workspaces, caches, evidence, gogcli state, SSH/AWS identity, and the yesod-control home
/var/lib/yesod/repos/yesod-aicoe	canonical checkout for the Yesod lane
/var/lib/yesod/repos/scr-wrap	canonical checkout for the scr-wrap lane
/var/lib/yesod/repos/yesod.work	canonical checkout for the yesod.work lane
/var/lib/yesod/dev5/{yesod-aicoe,scr-wrap,yesod.work}	manifested Beads mutation workspaces with canonical project identities
/var/yesod-workdir	recoverable dispatcher worktrees and run transcripts
/etc/yesod	root-controlled manifests, secrets, and explicit service-enable capabilities
/opt/yesod	pinned Node 22.23.2 and bd 1.1.0 host runtimes
/usr/local/bin	managed bd symlink plus reviewed static sjbis, gog, and sjbis-gog executables
/usr/local/libexec/yesod	host-level fleet telemetry helper
/mnt/nas/yesod-blobs	persistent content-addressed evidence mounted from homestar

The non-login yesod-control account runs the controller and most supporting services. The codefactory dispatcher deliberately remains under the existing stephen runner identity because its model credentials and worktree ownership were already scoped there. Secrets were copied only into root- or service-owned mode-0600 files; no credential entered Git.

Two release symlinks prevent an operational hardening branch from becoming an unreviewed merge controller. ops-current can stage an exact auxiliary commit needed for cutover, while current remains pinned to reviewed main until a promotion decision.

E.8.10. Standing Services Moved as a Set

The migration moved more than the controller. The active systemd set on `yesod-dispatch` is:

- `yesod-refinery-controller.service`;
- `yesod-gate-pool-proxy-8100.service`;
- `yesod-gate-fleet-poller.service`;
- `yesod-codefactory-dispatch.service`;
- `yesod-beads-collect.timer`;
- `yesod-archive-sweep.timer`;
- `yesod-status-push.timer`;
- `yesod-bridge-sjbis.timer`;
- `yesod-sjbis-gog.service`; and
- `mnt-nas-yesod\x2dblobs.mount`.

The units use systemd sandboxing, explicit enable files, immutable entry points, and bounded writable paths. The controller additionally requires the separate active capability; installing or enabling its unit alone does not grant merge authority. The legacy supervisor unit is absent.

The Beads collector was changed to read server-backed readiness directly from SQL. Later terminal-projection acceptance added separate, manifested repository-local `.beads` control workspaces under `/var/lib/yesod/dev5`; those exist for mutation identity, while claimability collection intentionally remains PostgreSQL-authoritative. Label collection was reduced to bounded server-native queries. Auxiliary one-shots were pointed at `ops-current`, while the long-lived merge processes remain on `current`.

Status reporting now probes Linux systemd state rather than workstation PID files. Its snapshots identify `yesod-dispatch`, allowing the old `pompom` status rows and readiness-stale warnings to be retired instead of blended into one fictitious host.

E.8.11. Human Communication Became Standing Infrastructure Too

Two easily confused bridges now have explicit ownership:

- `yesod-bridge-sjbis.timer` is the Yesod human-authority mail bridge. It consumes the human channel and asks the sjbis daemon on `dertog`.
- `yesod-sjbis-gog.service` polls Gmail and Google Chat, classifies genuine questions, submits them through sjbis, and routes approved answers back.

The Conductor runs static, checksummed Linux builds of `sjbis` and `sjbis-gog`, plus official gogcli 0.34.1. Minimal OAuth/keyring material lives under `/var/lib/yesod/.config/gogcli`; the Fireworks key, keyring settings, and explicit gog account selector live in `/etc/yesod/sjbis-gog.env`. Gmail and Chat read-only preflight passed as `yesod-control` before the Mac singleton was stopped.

No standing OpenCode server was moved to the Conductor. Durable Yesod mail is PostgreSQL-backed and host-independent, but OpenCode wakeups and live session transport

remain runtime-local. Centralizing one OpenCode server would only move the coupling. The host-neutral wakeup/adaptor problem remains documented in `ys-yes-53ok`.

One separate sjbis privacy issue also remains: message details are passed to a child `sjbis ask` process in its argument vector, making them locally visible to process inspection. That bug is tracked as `ys-sjb-bhgx`; it does not change refinery correctness, but it is part of the security boundary this wave made visible.

E.8.12. Cutover State Was Reconciled, Not Waved Away

The activation audit found fifteen `awaiting_merge` notes outside active v2 intake.

- One Yesod branch was already an ancestor of main. Its exact commit was backfilled and its note completed.
- Seven verified Yesod branches had their metadata repaired, known pre-cutover flaky counters cleared where appropriate, temporary holds removed, and deterministic v2 attempts created.
- Seven branches for other projects were verified on origin but kept under explicit holds pending reviewed v2 lanes. Their old "cutover in progress" labels were replaced with the true reason: production intake currently covers only `yesod` and `scr-wrap`.

This is an important definition of "empty." An empty v2 queue means no work is waiting in configured lanes. It does not mean every project in the catalog has been onboarded.

The resumed Yesod lane then proved real production behavior. A gate for `ys-yes-6bdu` ran across eight MicroVMs, reported 6,164 tests with zero failures, pushed the candidate, and completed the note. Subsequent re-gates continued from the reconciled queue without reviving the old launchd supervisor.

E.8.13. Two Final Alerts Improved the Design

The live dashboard initially reported `attempt_heartbeat_stale` even though the controller was healthy. The lease lasted 1,200 seconds and renewed 360 seconds before expiry, so the scheduler updated the attempt heartbeat only about every 840 seconds. The projection's freshness threshold was 60 seconds. Both components were internally consistent and jointly wrong.

Commit `5bddfd9d` added a fenced heartbeat-only update on every owned scheduler tick. It does not extend the lease expiry. The regression test advances time past 65 seconds and proves the heartbeat remains fresh, the lease deadline is unchanged, and the projection does not invent a stale attempt. The Conductor branch merged at `d434f0aa`, and the fix is live; subsequent gates showed a fresh advancing heartbeat without extending their lease deadlines.

The second alert was real. Two successful gates could not persist `provider-status.json` because `yesod-dispatch` inherited the Linux default `/mnt/nas/yesod-blobs`, but the NAS was not mounted. Gate verdicts remained valid by design and two `gate_artifact_persistence_failed` operator items opened.

The correction installed the NFS client, enabled a persistent systemd mount for `192.168.0.123:/volume1/blobs`, proved that the path was actually NFS, and performed write, checksum, atomic-rename, and byte-for-byte read-back probes. The provider still retained both terminal reports. They were fetched, written to content-addressed Blob storage, read back, and referenced in the audited operator resolutions. The inbox returned to zero open items.

E.8.14. A Third Lane Proved the Whole Terminal Boundary

Onboarding `yesod.work` turned the last abstract invariant into a production test. Its tool prefix is `ysw`, so new notes are `ys-ysw-*` and linked root Beads are `ysw-*`; three earlier provisional `ys-yes-*` notes were superseded by correctly identified replacements. The reviewed lane is `lane-v1-574981bc0bca351706b28b89`, uses Node 22.23.2 on `yesod-dispatch`, and runs exactly:

```
npm ci
npm run check
npm run build
```

The first green sentinel exposed why "gate green" is not the factory's terminal boundary. Its candidate reached `main`, but the root Bead remained open. The investigation found three independent host-boundary bugs:

1. terminal note projection committed before attempting Bead synchronization, but the synchronization result was discarded;
2. the service identity lacked a manifested `yesod.work` mutation workspace with the database's canonical `project_id`; and
3. `yesod-dispatch` still ran bd 1.0.4, whose dependency query was incompatible with the current shared Dolt schema even though bounded reads succeeded.

The fixes landed in stages. `aacbbf4d` moved root-Bead synchronization after the terminal database commit. `032e1372` added the manifested `yesod.work` workspace, contained host-path failures at the adapter boundary, and documented service-identity provisioning. The canonical workspace control files were installed mode 0600 under mode-0700 `.beads` directories, and bd was upgraded to v1.1.0. A checksum-pinned installer now owns that runtime at `/opt/yesod/beads-v1.1.0-linux_amd64`.

The final sentinel was `ys-ysw-6kot` / `ysw-p9l`. Candidate `321a7275` passed the local lane, reached `yesod.work` main, projected the note to `done`, removed the legacy queue row, and closed the root Bead automatically. This proved task identity, gate execution, push, terminal projection, and cross-host Beads mutation as one chain.

One final portability pass then changed shared workspace persistence from raw host paths to `~/dev5/...`, including migration of existing manifested rows. That prevents a Linux service registration from overwriting a Mac client's workspace with `/var/lib/yesod/...`, or the reverse. The same pass records the canonical `metadata.json.project_id` as durable database identity rather than disposable machine state.

The refinery itself enforced the final correction. The first portability candidate went red because two MicroVM assertions still encoded absolute host paths; its failing shard otherwise

reported 760 passes. After those assertions were corrected and existing manifested rows gained automatic migration, the successor candidate `86b3dc46` ran on eight distinct MicroVMs. The provider reported 6,258 tests collected, 6,231 executed, zero failures, zero retries, and zero infrastructure errors, then pushed the exact green candidate to `main`.

Promotion used the durable admission fence: revision 22 paused to 23, the immutable release was installed and verified, `/srv/yesod/current` advanced atomically, and admissions resumed at revision 24. The proxy, controller, and dispatcher all returned active with zero restarts; the active scheduler then reported a fresh heartbeat, zero tick errors, three idle lanes, an empty queue, and no operator items.

E.8.15. Onboarding Became an Idempotent Command

The successful third lane still depended on an operator remembering every place a repository had to exist. That was not a real onboarding procedure. A repository could have a catalog entry but no runner workspace, a workspace but the wrong note namespace, or a lane row that the active controller did not own.

Wave 8 therefore ended with two explicit commands:

```
yesod onboard --github https://github.com/OWNER/REPOSITORY
yesod onboard REPO_STUB status
```

The mutation converges catalog identity, note prefix, runner-workspace declaration, gate profile, controller manifest, PostgreSQL lane, runner provisioning, Conductor checkout, and active controller ownership. It takes its own exact-revision admission pause for lane registration. Repeating the real `yesod.work` command returned `changed=false`, `converged=true`, and eight `unchanged` steps; it did not duplicate any tool, workspace, profile, lane, checkout, or scheduler ownership.

Status proves nine named invariants: catalog identity, note namespace, workspace declaration, Beads workspace, dispatch-runner coverage, runtime lane declaration, refinery lane, controller ownership, and terminal acceptance. `yesod.work` returned both `verified=true` and `dispatch_ready=true`; all three runners resolved the same canonical server project identity, and the expected prefix remained `ysw`.

Two implementation defects were found by exercising the installed command, not its source-tree tests. First, a partially initialized Beads schema could return a fresh UUID without a metadata table in which to persist it. Empty workspace identity now requires both the issues and metadata tables before a canonical identity is minted. Second, a VCS/uv-tool installation looked for source-controlled manifests relative to `site-packages`. Release `e9a39099` made manifest discovery prefer the invoking source checkout or an explicit `YESOD_SOURCE_ROOT`, and mutating onboarding now refuses to write outside a reviewable checkout.

The completion audit found two more fail-closed gaps before the command was declared finished. The readiness model now requires the runtime declaration directly rather than relying on controller ownership to fail indirectly. Both runner provisioning and read-only status retry one transient routed Beads read once; a second failure remains visible. Convergence JSON

now separates `changed` from `converged`, and an incomplete non-dry-run exits one. Release `4f5a183b` passed 59 focused tests and an eight-MicroVM gate with 6,330 collected, 6,302 executed, zero failed, and 551.8 seconds of provider time.

The durable procedure is documented in repository onboarding.

E.8.16. The Refineries Page Became Self-Explaining

The redesigned page was authoritative but still assumed the operator knew its vocabulary. It also showed how many provider instance identities were linked to a gate without saying how many MicroVMs AWS actually reported running. Those are different facts: assignment is exact historical gate identity; fleet liveness is a separately timestamped provider observation.

Release `2d61e8f5` added the Conductor fleet poller's singleton PostgreSQL row to the same repeatable-read projection as lanes and schedulers. The browser still fetches only `/api/refineries/v2`; it does not race a legacy fleet endpoint or infer liveness from shard assignments. A prominent seventh summary card now says **MicroVMs running**, names AWS as the source, and shows observation age and freshness. Per-attempt rows instead say "N MicroVMs assigned to this gate."

Small keyboard-focusable question-mark controls now accompany summary metrics, scheduler policy, lane and attempt state, gate progress, assignment identity, freshness, and attempt-detail sections. Every explanation gives four things: meaning, what it blocks, authoritative source, and expected operator action. Escape, outside click, focus return, responsive placement, and complete-script parsing are covered by the browser contract.

The first production gate made the distinction visible in real time. The gate had eight assigned instance IDs while the first AWS fleet snapshot reported seven `RUNNING`; the next observation reported eight. Candidate `2d61e8f5` then completed green across eight distinct MicroVMs with 6,316 collected, 6,288 executed, zero failed, and a 537.8-second provider execution. The fleet returned to zero after push.

A post-deploy audit found one final inconsistency: the poller did not store the legacy endpoint's derived launch-storm tripwire, so v2 initially rendered an empty tripwire object. Follow-up `b1ec256c` derives the configured threshold from the authoritative running count itself. Its regression proves that 17 running instances trip the default threshold of 16 and degrade the page. That follow-up also passed an eight-MicroVM gate: 6,317 collected, 6,289 executed, zero failed, and 523.5 seconds.

The publish path produced one useful operational trap. Calling `deploy-dertog.sh` by absolute filename is insufficient because its payload paths are relative to the current working directory. Post-deploy API checks caught an initial no-op/stale publish. The correction ran the script *from inside* a fresh clone whose HEAD was the reviewed main SHA. Future promotions must preserve that working-directory invariant and verify a new API field, not only HTTP 200.

The final state is main and deployed release `4f5a183b`, admissions active at revision 38, a fresh zero-error controller heartbeat, a fresh fleet snapshot with zero running instances and a healthy threshold-16 tripwire, two idle dertog services, and all Conductor services active with zero restarts. The global operator CLI is pinned to the same release.

E.8.17. What Is Live

The live deployment is healthy and active:

Fact	Evidence
admissions	active at revision 38 after exact fenced promotion
controller	active on `yesod-dispatch`, zero restarts
old supervisor	unloaded and absent from launchd
merge ownership	lanes and attempts authoritative in PostgreSQL; merge lock free outside pushes
gate execution	Yesod remote eight-by-one; scr-wrap local one-by-one; yesod.work local exact npm check/build profile
operator inbox	zero open items after artifact recovery
artifact storage	active NFSv4 mount plus verified Blob round trip
visualization	public Refineries dashboard on `dertog` returns HTTP 200; contextual help and the fresh AWS `RUNNING` count are live
workstation	sixteen rollback plists preserved, zero relevant labels loaded, all persistently disabled

The source and deployment boundaries now agree. The Conductor migration, per-tick heartbeat, Blob inventory, third-lane profile, terminal write-back, workspace portability, idempotent onboarding, authoritative fleet visibility, and pinned host runtimes are all represented on main at `4f5a183b`. Promotion remains explicit—merging never silently retargets `/srv/yesod/current`—but the reviewed immutable release was advanced under the same admission fence and single-controller procedure used for activation.

E.8.18. The Exit Criterion Was Met

Wave 8 has already crossed the operational threshold that justified it: the factory no longer depends on a logged-in workstation or an LLM supervisor to interpret refinery state. A durable attempt owns every active gate, a stale worker cannot write past its fence, and the test workload runs on isolated machines rather than the client Mac.

The final code exit was deliberately narrow and is now satisfied:

1. `refinery-conductor-qii1` passed its normal gate and landed at `d434f0aa`;
2. `yesod-dispatch` promoted the merged controller, and per-tick heartbeat evidence stays fresh without extending lease expiry;
3. ordinary terminal gates persist evidence through the mounted Blob path;

4. the production boundary is explicitly three lanes, with other projects admitted only through the idempotent onboarding/status contract;
5. terminal acceptance now includes automatic root-Bead closure, not merely a green gate and Git push;
6. the Refineries page distinguishes actual AWS fleet liveness from gate assignment identity and explains its operational vocabulary in place; and
7. main, the Conductor release, dertog API/UI, and the globally installed CLI converge on `4f5a183b`.

The sjbis argument-vector exposure and host-neutral OpenCode wakeup gap remain visible follow-ups. They do not weaken the refinery exit, and they were not allowed to disappear into its success story.

The durable lesson is that determinism is not synonymous with centralization or serialization. The controller is deterministic because identity, authority, deadlines, and effects are explicit. The gate is fast because execution is parallel across isolated MicroVMs. The host map is safe because standing services live on a boot-managed Conductor, visualization lives on a read-oriented box, artifacts live on durable storage, and the workstation is free to be a workstation again.

E.9. Wave 9: Universal Intake and the Project Fleet

E.9.1. Wave Summary

Wave 8 proved that the deterministic refinery could safely own three production lanes. Wave 9 answered the harder operational question: could a project become a full Yesod citizen through one idempotent command, and could the fleet prove that every intended repository really had both a dispatch path and a merge path?

The answer at this checkpoint is yes. The reviewed fleet policy classifies 25 repositories as managed and eight catalog entries as explicitly excluded. All 25 managed projects have a unique note namespace, a canonical server-backed Beads identity, runner coverage, a deterministic repository lane, live controller ownership, and a terminal acceptance attempt. The fleet audit fails closed when a repository is missing from policy or any one of those boundaries drifts.

This wave also exposed three failures that a declaration-only audit would have missed. A Linux gate for `sjbgtd` followed a macOS-resolved machine-learning lock into multi-gigabyte CUDA packages and exhausted the Conductor's 20 GiB root disk. Later, successful sentinel merges could update PostgreSQL while failing to close their linked root Beads because the controller lacked repository-local routing metadata. Finally, the bridge lane declared Ruff as its gate but did not install the repository's locked development extra, so a fresh gate failed before lint began. These failures became versioned invariants, not operator folklore.

The final placement model remains intentionally strict:

- `pompom` is an interactive client workstation, with no standing factory infrastructure;
- `seykhl` is the Proxmox boundary and runs no Yesod application code directly;

- the `yesod-dispatch` guest is the Conductor and owns standing dispatch, refinery, bridge, polling, and collection services;
- `dertog` is the read/visualization box and cannot claim or merge work;
- `homestar` stores durable content-addressed artifacts; and
- AWS Lambda MicroVMs exist only for a remote gate lifecycle and its short, bounded idle-reap window.

E.9.2. The Starting Point Was Three Proven Lanes, Not a Fleet

The three-lane activation was real and healthy: `yesod`, `scr-wrap`, and `yesod.work` had all crossed their terminal boundaries. But an empty refinery dashboard only described those configured lanes. It said nothing about the other repositories already present in the catalog.

The immediate `yesod.work` defect made the distinction concrete. Notes had previously been created under `ys-yes-*`, the namespace owned by Yesod itself, instead of `ys-ysw-*`. Fixing the prefix was necessary, but a prefix alone did not provide a runner checkout, a Dolt workspace, a deterministic gate, a controller checkout, or a serialized merge lane.

Wave 9 therefore treated onboarding as a whole-system convergence problem. The unit of readiness was no longer "a tool exists in the catalog." It was "a note can be created, claimed, implemented, gated against an exact candidate, merged, projected terminal, and traced back to the correct root Bead."

E.9.3. Fleet Membership Became Reviewed Policy

`config/onboarding-fleet.yaml` is now the fail-closed inventory. Every catalog entry with repository metadata must have one reviewed disposition:

Disposition	Count	Meaning
managed	25	Yesod owns dispatch, runner provisioning, and a deterministic merge lane
excluded	8	the entry is deliberately outside factory ownership and carries a durable reason
unclassified	0	required for a green fleet audit

The managed set is:

`alphatok-cost-opt`, `als`, `amplifier`, `aysp`, `chunkpdf`, `clip-together`, `csync`, `dadtool`, `engage-watch`, `hnalg`, `instasort`, `mvr`, `nbgen`, `ppg-pdf`, `scr-rename`, `scr-wrap`, `shup`, `sjbgtd`, `sjbis`, `submgk`, `substack-sjbg-bridge`, `thmb`, `ttrac`, `yesod`, and `yesod.work`.

The excluded set is equally deliberate: `canva`, `gascity`, `gastown`, `gogcli`, `hyperframes`, `local`, `media-tooling`, and `synapse`. They are SaaS identities, local sentinels, upstream-owned repositories, or team-owned legacy workspaces. An exclusion does not mean "forgotten"; it means "Yesod is not authorized to own this repository's merge lane."

This closes a dangerous semantic hole. A repository newly added to the catalog does not quietly appear ready. Until policy, workspace, and lane declarations are reviewed, `yesod onboard status --all` reports drift and exits nonzero.

E.9.4. Onboarding Became One Idempotent Operation

The human-facing contract is small:

```
yesod onboard --github https://github.com/OWNER/REPOSITORY
yesod onboard REPO_STUB status
yesod onboard status --all --jobs 8
```

The mutating command converges canonical identity, note prefix, workspace manifest, coding-runner profile, deterministic merge profile, durable lane, all eligible runner hosts, Beads event schema, controller merge checkout, controller Beads route, and live controller ownership. Source manifests remain review authority: the command refuses to invent an unreviewed permissive gate.

The status command is read-only. Dispatch readiness now requires these nine pre-terminal invariants:

1. `catalog_identity`;
2. `note_namespace`;
3. `workspace_declaration`;
4. `beads_workspace`;
5. `dispatch_runner_coverage`;
6. `controller_beads_workspace`;
7. `runtime_lane_declaration`;
8. `refinery_lane`; and
9. `controller_lane_ownership`.

`terminal_acceptance` is the tenth and final verification invariant. This distinction is useful: a new project can be structurally dispatch-ready before its bounded sentinel has proved the complete terminal boundary, but it is not reported `verified` until that proof exists.

Replaying onboarding is safe. Correct state reports unchanged; drift is either repaired from reviewed authority or refused with a bounded explanation. The command never uses a workstation-specific path as standing controller authority.

E.9.5. Coding Gates and Merge Gates Are Different Contracts

The coding runner and deterministic refinery serve different purposes. A runner profile answers whether an agent's work is ready to hand off. The merge lane answers whether one

exact candidate may reach a target branch. They may use different commands and they live under different authority.

Wave 9 removed legacy client `repo_dir` values from runner profiles and made host-portable workspace identity explicit. The server-side Dolt `metadata._project_id` is canonical; machine-local metadata is a cache and guard, not permission to create a new database identity.

The merge fleet deliberately uses two provider shapes:

Shape	Lanes	Reason
eight isolated AWS Lambda MicroVM shards	1 (`yesod`)	the 6,000+ test suite benefits materially from fan-out and isolation
serial local Linux gate on `yesod-dispatch`	24	the project suites are small enough that MicroVM startup and per-repository key/IAM surface would cost more than it saves

"Local" here does not mean "on the operator's laptop." It means the gate runs under the restricted Conductor identity in an attempt-scoped worktree. Every repository and target branch remains serialized even though independent lanes can progress concurrently.

E.9.6. Twenty-Two Sentinels Tested the New Lanes

The three original lanes already had terminal evidence. Wave 9 created one bounded sentinel for each of the 22 newly managed lanes. Every sentinel branch contained an audited empty commit on that repository's current `main`: no feature content, only a unique candidate that could exercise intake, gate, compare-and-swap push, note projection, queue cleanup, and root-Bead closure.

Nineteen passed on their first attempt. Three—`sjbgtd`, `sjbis`, and `substack-sjbg-bridge`—went red during one shared disk-pressure incident. The red evidence was preserved. No result was relabeled green, and no branch was merged around the refinery.

After disk recovery, `sjbgtd` passed under its versioned Linux profile and `sjbis` passed from an audited empty retry commit. A fresh bridge retry then exposed a separate profile error: the v1 command invoked Ruff without installing the `dev` extra that contained it. Profile v2 repaired that exact contract and the preserved sentinel subsequently completed terminal.

This was precisely why the sentinel sweep existed. A manifest comparison would have declared the fleet ready without ever exercising dependency resolution, disk consumption, the provider boundary, or terminal write-back.

E.9.7. A macOS Lock Was Not a Safe Linux Gate

The `sjbgtd` gate initially synchronized the repository's full project lock on Linux. That lock includes `sentence-transformers`; on the Conductor it selected CUDA-enabled PyTorch and NVIDIA wheels even though the deterministic offline tests never exercise the real embedding model. Alongside a Rust target tree from `sjbis`, the download exhausted the 20 GiB root disk. The bridge lane then failed collaterally.

Recovery was bounded and evidence-preserving:

- exact failed worker PIDs were proven and stopped;
- only terminal attempt worktrees and the known failed Cargo target were removed;
- the UV cache was pruned, reclaiming about 2.1 GiB;
- no gate log, PostgreSQL attempt, or Blob artifact was deleted; and
- retries waited until no active attempt owned those paths.

The reviewed `sjbgtd` profile became `pytest:sjbgtd-linux-offline-v2`. It uses a lock-independent Python 3.14 environment containing only the lightweight runtime dependencies needed by the deterministic suite. It runs 371 offline tests and explicitly leaves three environment boundaries outside the merge gate: the stale Supabase client contract, the live self-hosted stack, and the cached Hugging Face model smoke.

This is not a permissive quarantine. The exact excluded files and the reason for each are versioned. A regression test forbids the merge profile from silently returning to the project lock or a CUDA dependency graph.

E.9.8. A Declared Linter Is Not an Installed Linter

`substack-sjbg-bridge` already locked Ruff in its optional `dev` extra, but the v1 merge command ran `uv run ruff` against only production dependencies. The provider correctly returned red with `ruff: No such file or directory`; this was a profile defect, not a project regression.

The reviewed profile became `lint:substack-sjbg-bridge-critical-v2` and now runs both commands with `uv run --locked --extra dev`. The exact corrected command passed against the preserved failed candidate before the source change entered the normal Yesod MicroVM gate. That source gate reported 6,322 passed, 28 skipped, and zero failed across eight distinct guests. After release promotion, the unchanged bridge candidate received a new immutable provider identity through the versioned profile, passed, reached `main`, cleared its queue row, and automatically closed root Bead `ssb-9l5`.

E.9.9. Terminal Projection Needed Its Own Controller Workspace

The first 19 green sentinels found a second boundary. PostgreSQL notes became terminal and branches reached `main`, but controller logs showed that several linked root Beads remained open. The controller had clean merge checkouts under `/var/lib/yesod/repos`, while the Beads router correctly looked for host-portable workspaces under the service home. Only the original three lanes had those routing workspaces.

The fix preserves the separation instead of cloning every repository twice. For each managed project, onboarding creates a lightweight workspace at:

```
/var/lib/yesod/dev5/<repo-dirname>/.beads
```

It contains reviewed server metadata, config, and the Dolt port—no second source checkout and no `.env` password. The protected service-level `dolt.toml` retains credentials. The canonical project ID is resolved from the Dolt database, existing database or project-ID conflicts fail closed, files are owned by `yesod-control` with restrictive modes, and the command proves `yesod beads info TOOL --json` under the controller's actual environment.

`controller_beads_workspace` is now a dispatch-readiness invariant. This makes the terminal write-back route part of every future onboarding, and the final sentinel proves not only a green gate and merge but the root-Bead closure that operators actually depend on.

The 19 notes that had already completed before this route existed were reconciled individually only after their terminal attempts and missing root closures were verified. The final three recovered sentinels—`sjbgtd`, `sjbis`, and the bridge—then closed their roots automatically through the new route.

E.9.10. Placement Did Not Drift During Expansion

Universal project intake did not turn the workstations or hypervisor into a server room.

pompom

The client keeps the global CLI, source checkouts, interactive credentials, and user-started Codex/OpenCode/SSH processes. It runs no standing controller, runner daemon, refinery supervisor, gate poller, bridge, status publisher, OpenCode server, or Yesod web service. Preserved launch-agent plists remain disabled rollback material, not active infrastructure. The bare `yesod` command resolves through `~/.local/bin/yesod` to the stable `~/yesod-services-clone` venv at canonical main; it is not a frozen package snapshot and does not execute from Stephen's mutable development checkout.

seykhl

The Mac Studio is the Proxmox node. It owns VM 112's compute and disk boundary, but no `/srv/yesod`, `/var/lib/yesod`, application credential, or Yesod service belongs on the hypervisor. The applications live inside the guest.

yesod-dispatch

The Conductor guest owns the deterministic controller, an operator-only codefactory daemon, loopback MicroVM proxy, fleet poller, collectors/timers, service-health publisher, human-authority bridges, immutable releases, 25 clean merge checkouts, and 25 lightweight Beads

routing workspaces. The codefactory daemon is alive for management and maintenance but has `operator: true`, so it never claims coding work; the three Linux coding runners remain the worker tier. Global refinery-lane capacity is two. The controller is mechanical; it does not host an LLM decision loop.

`dertog`, `homestar`, and MicroVMs

`dertog` serves the public dashboard and read APIs. It can display PostgreSQL and fleet projections but has no merge capability. `homestar` exposes the NFS artifact store and interprets no factory state. MicroVMs receive one exact gate candidate and disposable credentials, then disappear; they retain neither authority nor durable evidence.

E.9.11. Interim Disk Headroom Without a Placement Change

The sentinel sweep made the Conductor's 20 GiB root-disk limit operationally visible: one clean Rust build took the guest to 90% utilization even after bounded cache cleanup. At a quiescent boundary, VM 112's thin-provisioned disk was expanded online on `seykhl` from 20 GiB to 64 GiB; partition 1 and ext4 were grown in place without rebooting or moving a service. The final snapshot showed roughly 45 GiB free.

This is explicitly interim headroom while a much beefier host is forthcoming. It does not authorize Yesod code on the Proxmox host and does not collapse the Conductor, builder, or sandbox roles. The release was staged as a shallow exact canonical commit from the trusted workstation because the interactive guest account intentionally has no GitHub deploy key; no new standing credential was added to `yesod-dispatch`.

E.9.12. The Operational Commands Are Boring on Purpose

For a new Yesod-owned repository:

```
yesod onboard --github https://github.com/stephenVertex/NEW-REPO
yesod onboard NEW-REPO status --json
```

For the whole reviewed fleet:

```
yesod onboard status --all --jobs 8 --json
```

The fleet is not universally ready unless that last command reports:

```
managed=25
dispatch_ready=25
verified=25
excluded=8
unclassified=0
fleet_ready=true
```

The public operator surface remains:

```
https://dertog.tailb4b58.ts.net/yesod/viz/refineries.html
```

Its queue describes the configured fleet, not the catalog in the abstract. Question-mark help defines the operational fields, and MicroVM fleet state is reported separately from lane state so "one active lane" is never mistaken for "one running guest."

E.9.13. What Is Live

- Canonical GitHub main and `/srv/yesod/current`: `958326e7fe42eb852a95983f86a696fc81e10f1e`.
- Active scheduler: `global-refinery-v2`, controller version `refinery-controller-v1`, PID 576980, 25 configured lanes, global capacity two. The operator-only codefactory daemon is PID 608904 from the same release.
- Admissions: active at revision 90, with a fresh heartbeat and zero tick errors.
- Managed fleet: 25 dispatch-ready and 25 terminal-verified.
- Exclusions: eight reviewed, zero unclassified repositories.
- Client CLI: bare `yesod` resolves from `~/yesod-services-clone` at `958326e7`; the fleet-wide `onboard --all --jobs` surface is available.
- Providers: one eight-shard MicroVM lane and 24 serial local Conductor lanes.
- Final source gate: eight distinct MicroVMs, 6,322 passed, 28 skipped, zero failed; the final idle snapshot reports zero running MicroVMs and a healthy idle proxy.
- Queue and work: merge queue empty, no active refinery attempts, and no running dispatches at the final verification snapshot.
- Dispatcher health: `yesod-dispatch` is a fresh `operator` runner with zero claims. Its operational probe is anchored to the lightweight `yesod` Beads route; the inherited workstation-era `clip-together` probe was removed because that source checkout intentionally does not exist on the Conductor.
- VM 112 root disk: 64 GiB provisioned, about 45 GiB free after online growth.
- Public dashboard: HTTP 200 from `dertog`.

E.9.14. The Exit Criterion

Wave 9 is complete only when "you can file a task for any managed Yesod project" has one unambiguous meaning:

1. the note is created in that project's unique namespace;
2. an eligible runner can claim it against the canonical Dolt workspace;
3. the worker's branch enters the correct repository/target lane;
4. the reviewed deterministic gate runs on the intended provider;
5. only the exact green candidate may reach `main`;
6. PostgreSQL, the queue, and the lane record the terminal result; and
7. the linked root Bead closes through the controller's own routing workspace.

The fleet audit and sentinel ledger now test that statement rather than asking an operator to remember it. Future repositories still need a reviewed policy entry and gate; "universal"

means every intended managed project follows one repeatable onboarding contract, not that Yesod silently assumes ownership of every URL in the catalog.

E.10. Wave 10: Lifecycle v2 and the Self-Governing Factory

E.10.1. Wave Summary

Wave 9 made every reviewed repository a factory citizen. Wave 10 changed what happens before execution: a raw thought is no longer treated as approved work, planning is no longer a transient agent convention, and an approved plan no longer depends on a mayor or ephemeral supervisor remembering to enqueue it.

The production lifecycle now has three independent axes:

- lifecycle says where the work is;
- disposition says whether an orthogonal veto applies; and
- automation policy says whether the user wanted capture, planning, or delivery.

The main path is:

```
captured → planning → planned → open → claimed → in_progress
         → awaiting_verification or awaiting_merge → merging → done
```

`open` has one narrow meaning: the current revision is planned, approved, and eligible for deterministic dispatch when its remaining readiness predicates pass. It is not an inbox. Ordinary feature and bug creation is therefore safe capture; `--plan` asks for a validated plan and stops; `--auto-plan` asks the factory to deliver under standing policy.

The rollout is active in production at lifecycle control revision 21. The writer, planner, and scheduler are enforced; the implementation backlog has zero legacy notes; the factory's own canary reached `done`; its exact candidate passed on eight distinct MicroVMs; a formal rollback and replay were proved; and a fresh 15-check acceptance authorized activation. The deterministic refinery remains the only merge authority.

Boundary	Active result
source release	`046b44f0486acfc715a7bacae18471c283c90594`
lifecycle control	revision 21, `active`, enforced
canary	`ys-yes-1sqs`, root `yes-jx4e9`, terminal `done`/closed
coding run	`run-7cb684da37d1` on `yesod-runner-1`
source commit	`c00035ef147e2f410719b1eef4d97a478e89e59e`
refinery attempt	`attempt-intake-v1-4ca8d68b0af8b5e4e35dcf96c0de95c8`

Boundary	Active result
gate	`gate-exec-v1-6d067ba8337686a26db55f663aeb246c1d15bd2a`
gate proof	eight distinct MicroVMs, green, exit 0, 6,471 passed, 28 skipped
deployment effect	`effect-deploy-v1-24eafa5372a556d6d608c286a65a86b4`, confirmed applied
final acceptance	`lifecycle-acceptance-v1-156730450ade8ba19e089976`, 15/15
final refinery snapshot	26 idle healthy lanes, no queue, no operator items, zero running MicroVMs

E.10.2. The Problem Was That `open` Meant Too Many Things

Before this wave, a feature request or bug report normally arrived as `open`. That single word might mean "I just wanted to remember this," "please plan this," "a planner created some Beads," "I approve this," or "a runner may claim this now." The system had useful pieces—planner agents, Beads, dispatch lanes, runner claims, refinery intake—but no durable contract connected them.

This created two opposing risks. Capture-only ideas could look runnable, while work the user actually wanted delivered could sit outside the queue until a mayor, dispatcher agent, or ephemeral refinery supervisor noticed it. The first problem was excess authority. The second was missing mechanism.

The design separated the questions instead of adding another overloaded status:

Axis	Question	Examples
lifecycle	where is the work?	`captured`, `planning`, `planned`, `open`, `in_progress`, `awaiting_merge`, `done`
disposition	may automation act?	`active`, `held`, `blocked`, `needs_rework`, `quarantined`, `deferred`
automation policy	how far did the user authorize it?	`capture`, `plan`, `deliver`

A dispatch target remains routing metadata. It does not authorize planning, execution, credentials, deployment, or destructive side effects. Capacity, backoff, and dependency waits remain derived explanations rather than new states.

E.10.3. The CLI Became Explicit Without Becoming Fussy

The common cases are now intentionally small:

```
# Record intent. Do not plan or dispatch it.
yesod tool <tool> feature-request "..."
yesod tool <tool> bug-report "..."

# Produce a grounded, validated plan, then wait for approval.
yesod tool <tool> feature-request "..." --plan

# Plan, validate, approve under standing policy, and deliver.
yesod tool <tool> feature-request "..." --auto-plan
```

All three begin with durable capture. `--plan` and `--auto-plan` write policy; they do not synchronously ask one process to perform the entire pipeline. This is why the workflow survives restarts. The planning reconciler, promoter, dispatch reconciler, runner, and refinery can each stop after one auditable step and resume from PostgreSQL.

E.10.4. Plans Became Revision-Bound Durable Objects

A root Bead is useful grounding, but its mere existence is not proof that a planner processed the current request. Wave 10 binds planning to immutable input:

1. material intent edits increment `note_revision`;
2. one planning job exists for `(note_id, note_revision)`;
3. the artifact records plan identity, provenance, validation, and a fingerprint;
4. `planned_revision` must equal `note_revision` before promotion or claim;
5. an edit invalidates the old plan rather than silently changing live scope; and
6. a dispatch arm is bound to the note revision, plan fingerprint, project lane, repository, target branch, and model target.

Planner output is stored durably, including bodies too large for bounded relational fields. Large payloads are hydrated through content-addressed Blob references, so retry and restart do not depend on a vanished provider response or one agent's context window.

The planner controller uses stable provider-job identity, a lease, bounded retry state, and a deadline. A provider that still reports `running` after the deadline is not allowed to keep the job pending forever: the controller marks the deadline condition and cancels or escalates through durable state.

E.10.5. Deterministic Promotion and Dispatch Replaced Remembering

The lifecycle controller owns state movement; agents own semantic judgment.

- The planning reconciler turns eligible captured notes into one durable job.
- The validator accepts only an artifact for the exact current revision.

- The promoter moves `planned` to `open` only after explicit approval or the `deliver` policy passes every risk and readiness guard.
- The dispatch reconciler turns `open + active` into one lane-scoped arm and claim when onboarding, dependencies, budget, routing, and capacity agree.
- The runner establishes the run/worktree/provider effect and produces a verified candidate branch.
- The refinery alone gates, pushes, deploys, and terminalizes it.

This is the answer to whether an ephemeral refinery supervisor is still required: no agent is required to keep the pipeline moving. A mayor remains a useful product and coordination role. A planner remains necessary for semantic decomposition. Neither is a liveness mechanism or merge authority.

E.10.6. Implementation Was a Program, Not an Enum Patch

The active implementation includes:

- additive schema and compatibility projections;
- centralized audited compare-and-swap writers;
- durable lifecycle control revisions and enable-token fencing;
- capture/plan/deliver CLI and API semantics;
- planning jobs, leases, provider recovery, retries, deadlines, and artifacts;
- validator and auto-promoter policy;
- revision- and lane-bound dispatch arms;
- deterministic intake from approved `open` work;
- recovery and invariant diagnostics;
- lifecycle API and all-notes state-diagram filters;
- contextual help explaining state, authority, blocking impact, and operator action;
- activation acceptance bound to a specific control revision; and
- a non-destructive rollback/replay procedure.

Focused lifecycle, broker, scheduler, runner, runtime, and UI tests established internal coherence before the production canary. The complete refinery gate then tested the exact constructed candidate, not merely the worker's source branch.

E.10.7. Production Found the Bugs Unit Tests Could Not

The staged cutover was deliberately allowed to discover boundary failures before full authority was granted.

Source branch and target branch were conflated

The canary initially exposed a resolver that treated the feature source branch as though it were the integration target. The durable arm now carries both identities, and resolution rejects ambiguity. A note can explain exactly which branch supplies work and which branch the refinery may update.

Identity had to include the project lane

Two repositories may contain the same Git SHA and gate profile without being the same operation. Global idempotency rejected the second lane. Gate and dispatch identity now includes the repository/target lane; a SHA is never treated as a globally unique project identity.

Recovery needed an atomic claim fence

Planner and dispatch recovery could otherwise observe a durable object and race to own it. The claim path now binds the current revision, artifact, lane, and owner under one fenced transition. Diagnostics distinguish "no work," "not ready," "already owned," and "contradictory evidence."

Retry publication and Blob hydration were part of correctness

A retry that only changes a private provider row is invisible to the lifecycle projection. Retry state is now published durably, and artifact bodies are rehydrated before validation or dispatch. Restart behavior therefore matches the ordinary path.

A stale historical branch could poison a newly armed note

Arming a new revision must not reuse an earlier branch merely because the note id is stable. New arms discard historical branch assumptions and bind only the current plan and source facts.

Acceptance needed exact gate invariants

The first production implementation verified that a gate existed and was green. The strengthened acceptance requires the expected provider, terminal green, exit code zero, and eight distinct MicroVM instance identities. This turned the user's throughput and isolation expectation into an activation condition.

Closed Beads could disappear behind a large open backlog

The terminal-close acceptance initially failed even though the canary root was closed. The standing collector requested only 100 issues, and more than 100 open Yesod issues sorted ahead of the newly closed root. Production currently overrides `yesod-beads-collect.service` to use `--issue-limit 1000`. The source-level follow-up is to fetch linked terminal roots completely by construction, then remove the host override only after that fix is deployed.

Public URL stability was an operational invariant

The application page existed at `/yesod/viz/refineries.html`, while activation acceptance required a stable `/yesod/refineries` surface. `dertog` now serves the stable alias through its Nginx router and retains the historical path for compatibility.

E.10.8. The Cutover Was a Sequence of Capabilities

The lifecycle enable token is separate from service liveness. A running controller without the token cannot acquire write, planning, or scheduling authority. The token lives on `yesod-dispatch` at `/home/stephen/.config/yesod/lifecycle-v2-enable-token`, mode 0600; its contents are never part of an evidence bundle or documentation.

Activation proceeded through durable control revisions:

1. establish schema, readers, projections, and invariant reports;
2. enable audited writers;
3. enable planner/validator behavior;
4. restrict scheduling to one exact canary;
5. prove source/target resolution and revision-bound arm identity;
6. run the worker and enroll its verified branch;
7. construct and gate the exact candidate on eight MicroVMs;
8. confirm push, deployment effect, note terminalization, and root closure;
9. migrate the remaining legacy terminal implementation notes without dispatch;
10. pass the first activation acceptance;
11. perform a formal rollback with all writers disabled;
12. prove the heartbeat advances while durable lifecycle facts do not change;
13. replay migration and restore readers, writers, planner, and canary in order;
14. reject the old acceptance because it was bound to a stale revision;
15. pass a fresh 15-check acceptance; and
16. enter active revision 21.

The formal rollback was revision 15 to revision 16. Its durable fact digest remained exactly unchanged while the service continued to tick. Recovery advanced through revisions 17–20. The revision-15 acceptance was correctly rejected after recovery; activation used only the fresh revision-20 report.

E.10.9. The Canary Went Through the Real Factory

The canary was not an administrative state edit. It used the same boundaries as ordinary work:

```
ys-yes-1sqs revision 10
  → planjob-f6e9470c80dfa6a928b1c793
  → dispatch-arm-f7fda887a5241f8a3506276b
  → run-7cb684da37d1 on yesod-runner-1
```

```
→ source c00035ef147e...
→ attempt-intake-v1-4ca8d68b0af8b5e4e35dcf96c0de95c8
→ candidate 046b44f0486a...
→ gate-exec-v1-6d067ba8337686a26db55f663aeb246c1d15bd2a
→ eight distinct MicroVMs, green, exit 0
→ origin/main 046b44f0486a...
→ deployment effect confirmed applied
→ note done; root yes-jx4e9 closed
```

The gate reported 6,499 collected tests: 6,471 passed and 28 skipped, with zero failures. After terminalization the fleet reaped to zero running MicroVMs.

E.10.10. Legacy Migration Was Conservative

The controller found seven post-bootstrap implementation notes already known to be terminal through legacy evidence. It classified them through the v2 capture-terminal path, recorded a plan before applying it, and replayed the migration to prove idempotency. No legacy note became new dispatch work. The final inventory reports zero legacy implementation notes.

Historical bootstrap alerts required a similar evidence rule. Fourteen open `lane_not_configured_for_rec` items belonged to lanes that were later configured. Every linked attempt had subsequently reached `succeeded`, every lane was enabled and idle, and no current attempt or queue row existed. Only then were the items resolved with structured immutable explanations. The final operator inbox is empty; the history remains queryable.

E.10.11. What Lives Where Now

The placement model did not change during lifecycle activation; it became more strictly documented and re-audited.

Place	What lives there	What does not live there
`pompom`	interactive CLI, source checkouts, operator sessions, unloaded rollback plists	no standing Yesod, refinery, dispatcher, poller, bridge, dashboard, sjbis-gog, or OpenCode service
`seykhl`	Proxmox and VM 112's virtual hardware	no Yesod application process or application credential on the hypervisor

Place	What lives there	What does not live there
`yesod-dispatch`	lifecycle controller, deterministic refinery, gate proxy, fleet poller, operator-only dispatch maintenance, collectors/timers, bridges, release and merge checkouts	no public visualization and no coding claims
`yesod-runner-1`	active Linux coding-runner service and governed worktrees	no merge or deployment authority
`yesod-runner-2/3`	provisioned coding-runner tier when enabled by fleet policy	no merge authority
`dertog`	Yesod read APIs, dashboards, WebSocket watcher, Nginx routing, and `sjbis`	no planning claim, coding claim, gate launch authority, push, or merge authority
`homestar`	content-addressed NFS Blob storage	no interpretation of state
AWS MicroVMs	eight disposable isolated pytest guests while the Yesod gate is active	no durable catalog credential, state, or merge key

The deployment uses immutable source releases:

```
/srv/yesod/releases/<commit>
/srv/yesod/current       → active controller/dispatcher release
/srv/yesod/ops-current   → active auxiliary-service release
```

At the final snapshot both pointers, the coding runner, and the visualization deployment identify `046b44f0486acfc715a7bacae18471c283c90594`.

`pompom` was audited after its reboot. No matching launchd job was loaded and no standing lifecycle/refinery/dispatch process existed. Preserved plist files are rollback material, not running infrastructure.

E.10.12. The Operator Surface Shows Real Capacity

The Refineries page at `https://dertog.tailb4b58.ts.net/yesod/refineries` reads one coherent PostgreSQL projection. It separately displays:

- configured scheduler capacity;
- active, waiting, degraded, and operator-required lanes;
- queued and current attempt identity;

- gate progress and assigned provider instances;
- actual provider-wide AWS `RUNNING` count;
- fleet snapshot source time, age, TTL, proxy state, and consistency;
- a launch-storm tripwire; and
- contextual question-mark help for the operational vocabulary.

This distinction matters. One active lane may own eight MicroVMs, and eight assigned gate identities are not proof that all eight guests remain running. The dashboard exposes both facts without requiring browser-side AWS credentials.

E.10.13. Final State

The final production audit reported:

- lifecycle control revision 21, active and enforced;
- writer, planner, and scheduler enabled;
- zero legacy implementation notes;
- admissions active at revision 140;
- controller and runner heartbeats fresh;
- canonical and deployed release `046b44f0` everywhere in authority;
- 26 configured project lanes, all idle and healthy;
- zero queued attempts and no current attempt;
- zero open operator items;
- zero running MicroVMs, healthy idle proxy, consistent fleet snapshot;
- public refinery and legacy dashboard paths returning HTTP 200; and
- no standing Yesod infrastructure on `pompom` or directly on `seykhl`.

The rollout evidence is retained privately on `yesod-dispatch` under `/home/stephen/.local/state/yesod/v2-evidence/20260811T094900Z`. The directory and files are permission-restricted and hash-manifested; no enable token or database credential is part of the bundle.

E.10.14. The Exit Criterion

Wave 10 is complete when a user can choose intent explicitly and the factory can carry that intent without a resident agent remembering the next step:

1. ordinary creation captures safely;
2. `--plan` produces a validated revision-bound plan and stops;
3. `--auto-plan` progresses only through standing policy and deterministic guards;
4. `open` is the sole approved handoff to dispatch;
5. one lane-scoped arm and one runner claim own execution;
6. the refinery gates and merges only the exact candidate;
7. terminal evidence closes the note and governing root;
8. rollback disables authority without deleting history; and
9. restart/replay resumes from durable identity without duplication.

That criterion is now met. Semantic agents remain essential, but factory liveness no longer depends on an ephemeral supervisor, mayor, or workstation session. The system can explain who authorized each transition, which revision it used, what it is waiting for, which external effect occurred, and why the exact candidate was allowed onto `main`.

F Gas Town, Gas City, and Yesod

Yesod does not exist in isolation. Two closely related open-source projects—**Gas Town** and **Gas City**—attack many of the same problems: durable work outside an agent's context window, parallel coding sessions, role specialization, worktree isolation, recovery after session death, and coordination across repositories.

The names can obscure the relationship. Gas Town is the older, opinionated multi-agent workspace manager. Gas City extracts its reusable machinery into a role-agnostic orchestration SDK. A real `gastown` pack then recreates Gas Town's familiar organization on top of Gas City using configuration, prompts, formulas, orders, and scripts.

This appendix compares three product shapes rather than treating the names as interchangeable:

1. the standalone **Gas Town** system and its `gt` CLI;
2. the role-agnostic **Gas City** engine and its `gc` CLI;
3. the **Gastown pack** running on Gas City.

F.1. Research Snapshot

These projects move quickly, and their prose sometimes describes a design before the default execution path has caught up. The comparison therefore uses pinned source snapshots and treats code, current configuration, and executable pack definitions as stronger evidence than older design documents.

System	Snapshot	Date / release context
Yesod	`cf091d2e`	book snapshot, 2026-07-11
Gas City	`6c8dfdce`	`main`, 2026-07-13; stable `v1.3.4`
Gas Town	`f97ed211`	`main`, 2026-07-13; stable `v1.2.1`
Gas City packs	`04fc38f6`	`gastown` pack, 2026-07-13

Gas City is the strategic successor architecture, but Gas Town is not an abandoned repository or a completed in-place migration. Both remain active. The official command map warns that their architectures are not identical, and moving from `gt` to `gc` means re-expressing the operating model rather than upgrading one workspace database in place.

> **Terminology Trap — Gas City is not merely Gas Town 2.** Gas Town is a particular organization. Gas City is a kit for constructing organizations. The Gastown pack is one organization built with that kit.

F.2. The Central Distinction

The most useful comparison is not "which system has more agents?" It is **where each system draws the boundary between judgment and mechanism**.

Yesod is **vertically deterministic**. It codifies one delivery pipeline deeply: claim rules, worktree creation, process governance, proof of work, branch publication, merge selection, gate execution, push verification, landed-commit proof, and post-merge hooks. Roles remain flexible at the planning and diagnosis edges, but the path by which code reaches `main` is deliberately narrow.

Gas City is **horizontally deterministic**. It codifies reusable orchestration machinery: desired-state reconciliation, sessions, runtime providers, elastic pools, formula graphs, dependency gates, fan-out, retries, orders, events, and health patrol. It intentionally does not prescribe a universal planner, reviewer, merge authority, or delivery policy.

Gas Town is more **agent-centric**. Its daemon and CLI provide substantial mechanics, but Mayor, Deacon, Witness, Refinery, Polecat, and Dog sessions interpret long role guides and workflow formulas to run much of the organization.

Dimension	Yesod	Gas Town	Gas City
Product posture	opinionated autonomous factory	opinionated multi-agent workspace	orchestration-builder SDK
Roles	explicit primers plus fixed services	built-in named organization	zero roles in the core
Workflow	fixed evidence pipeline plus Beads DAG	convoys, molecules, role patrols	pack-defined formula graphs
Integration	deterministic AME	agent-run Refinery	pack-defined; no core policy
Extensibility	tools, processes, profiles, lanes	role and runtime configuration	packs, providers, formulas, orders
Primary state	PostgreSQL + Beads/Dolt + Git	town/rig Beads/Dolt + Git	pluggable Beads store + events

This is not a maturity ranking. A general SDK must leave choices open that a vertically integrated factory can make mandatory.

F.3. Where the Roles Live

Gas Town makes its organizational metaphor part of the product. The Mayor is the human-facing coordinator. Deacon patrols the town. Witness watches a rig. Refinery processes its merge queue. Polecats implement bounded assignments. Dogs perform maintenance and recovery. The directory tree, role identity, tmux sessions, and operating procedures reinforce the taxonomy.

Yesod also has named roles, but it separates them from the services that enforce factory mechanics. `yesod prime --planner` or `yesod prime --triage` loads an operating contract into a session. The runner, AME, gate proxy, and other services do not become those roles; they remain code with narrower state transitions.

Gas City deliberately hardcodes no role names. An agent is a prompt, provider, scope, work query, and session policy declared by a pack. A "planner" has no privileged meaning to the Go controller. The same engine can load a Gastown pack, a Ralph-style loop, a review swarm, or a project-specific organization.

The `gastown` pack proves that this is more than an aspiration. It defines Mayor, Deacon, Boot, Dog, Witness, Refinery, and Polecat agents; patrol and shutdown formulas; scripts; orders; commands; and tests. Crew and polecat become operating styles rather than platform types. A Gas City feature in the controller can therefore improve every pack without teaching the core what a Witness is.

> **Design Trade-off — Law versus configuration.** A hardcoded role system is easier to understand but harder to recombine. A role-free substrate is easier to extend but cannot, by itself, guarantee that a pack chose safe responsibilities.

F.4. Planning and Decomposition

All three systems put ambiguous planning primarily in agents.

In Yesod, a planner reads source and chooses merge boundaries. Each feature gets one note and root Bead; child Beads form its internal checklist, and note dependencies order separately mergeable features. The planner rates complexity and leaves the notes unarmed. The durable plan survives the session, and deterministic services later decide readiness and claimability after the dispatcher chooses lanes.

In Gas Town, the Mayor commonly turns intent into issues, convoys, formulas, and Polecat assignments. Planning conventions are supplied through the Mayor's role guide and workflow definitions.

In Gas City, planning is pack policy. A formula can encode a repeatable planning method, including parallel review legs and human gates, but an agent usually performs the semantic decomposition. Once a version-2 formula has materialized a graph, the controller—not the originating agent—drives its dependency and control structure.

That last distinction is important. Gas City formulas can contain deterministic **control beads** for checks, retries, fan-out, drains, tallies, and finalization. Agents execute work beads; the controller advances the graph outside any one session. This is more general than Yesod's

current fixed note-to-Bead dispatch path, although Yesod's path has stronger delivery-specific proof.

F.5. Claim, Execution, and Isolation

Yesod's runner polls for the intersection of an eligible note and a dependency-ready Bead, wins ownership through a short PostgreSQL transaction, assigns the Bead, creates a new Git worktree, launches the selected lane, enforces time and token budgets, and independently reconciles the result. The agent is not trusted to decide whether its own transcript proves progress.

Gas Town uses Beads hooks, sling operations, a capacity scheduler, and Polecat sessions. Work is commonly isolated in a Polecat worktree. `gt done` performs substantial mechanical completion work, but the larger workflow still assumes the Polecat follows its injected formula and completion discipline.

Gas City separates these concerns into providers and configuration. A controller reconciles desired agents to sessions, sizes pools from demand queries, and can run through tmux, subprocesses, ACP, Kubernetes, SSH, `exec:` providers, or other runtime packs. Worktree isolation is available through pack startup hooks and `work_dir`, but it is a pack choice rather than a universal platform invariant.

Gas City is consequently stronger as a reusable runtime layer. Yesod is stronger at making one execution contract mandatory for factory-dispatched code work.

F.6. Integration Is the Clearest Divergence

The sharpest difference is what happens after a worker says it is finished.

In Yesod, the runner proves that qualifying commits exist, executes declared acceptance checks, pushes a named branch, and verifies the remote branch. The AME then resets a dedicated checkout, verifies candidate branches, chooses a compatible batch, performs the merge, runs the authoritative gate, pushes `main`, verifies the remote SHA and commit ancestry, records the result, and invokes the configured post-merge hook.

The worker cannot turn its own assertion into a merge. The merge process does not need an LLM to remember the sequence.

Gas Town has a Go package containing deterministic merge primitives, claims, gates, and batch/bisect code. Its design documentation describes a Bors-style queue. At the inspected snapshot, however, the normal Refinery manager still starts an LLM agent in tmux, and the batch `ProcessBatch` path has no non-test caller. The current Refinery guide instructs the agent to select work, run Git commands, classify failures, decide whether a conflict is trivial, run tests, and push.

The Gastown pack on Gas City states the boundary even more directly: merge and conflict decisions belong to the Refinery agent, not to Go code. Its patrol formula contains the rebase,

test, rejection, branch cleanup, merge, push, and Bead-update procedures that the agent is expected to interpret and execute.

Gas City core does not define a universal merge queue because a generic orchestration SDK cannot assume every workflow produces Git branches. A pack can implement a deterministic executable order, an agent-driven Refinery, a pull-request workflow, or no code integration at all.

> **Invariant — Preserve the narrow door.** A Yesod integration with Gas City should use Gas City's runtime and graph machinery without moving AME authority into a pack prompt. The pack may request integration; deterministic Yesod services should continue to prove and perform it.

F.7. Recovery and Health

Gas Town's recovery system mixes code and agent judgment. The daemon monitors mechanical liveness and restarts important sessions. Deacon assesses town-level progress. Witness investigates rig work, orphaned Beads, and stuck Polecats. Dogs execute shutdown warrants. This permits nuanced recovery, but it also spends model capacity on operational loops and depends on agents correctly following long recipes.

Gas City moves more of that infrastructure into its controller. Health Patrol performs desired-state reconciliation, configuration fingerprinting, idle retirement, restart-window accounting, crash-loop quarantine, pool draining, orphan cleanup, bounded dependency-aware starts and stops, order dispatch, and wisp garbage collection without requiring a Deacon role.

The Gastown pack can still add Deacon, Witness, and Dog agents for work-layer questions: Is work actually progressing? Is a failure structural or transient? Is an apparently silent process performing a legitimate long tool call? The architectural improvement is that these agents advise above a deterministic supervision substrate rather than owning basic process existence.

Yesod sits between these models. Its runner and AME perform strong deterministic reconciliation around execution and integration. Its SPS and watchdog layers are less mature than Gas City's general session supervisor, especially around chronic degradation, cause-aware recovery, and cross-host actions. Gas City's crash quarantine, content-hash drift detection, provider abstraction, and bounded lifecycle waves are directly reusable ideas.

F.8. Parallelism

Gas Town parallelizes by slinging independent Beads to Polecats, potentially across rigs. A scheduler can cap concurrent Polecats so a large convoy does not exhaust API quota, memory, or CPU. Its default operational center remains a local town with tmux-backed sessions.

Gas City elevates parallelism into the formula engine. Independent graph steps become ready together, elastic pools expand to demand, review lanes fan out, drains wait for whole runtime-discovered sets, and the controller advances the graph outside the human's session. Runtime providers also make the execution location replaceable.

Yesod parallelizes ready Beads across eligible runner hosts and model lanes. It adds atomic claims, per-run worktrees, host affinity, and a separate integration queue. Its parallelism is currently less general than Gas City's arbitrary workflow graph, but more tightly coupled to verified code delivery.

The strategic difference is the unit of reuse:

- Gas Town reuses an **operating model**;
- Gas City reuses an **orchestration language and runtime**;
- Yesod reuses an **evidence-governed factory path** across tools.

F.9. Durable State and Evidence

Gas Town uses a two-level Beads architecture. Town-level Beads hold cross-rig coordination, identity, convoys, and mail. Rig-level Beads hold implementation issues, merge requests, and workflow state. A shared Dolt SQL server provides versioned storage; Git branches and worktrees hold code.

Gas City makes Beads the universal work substrate. Tasks, sessions, messages, convoys, formula runs, and control state are different typed Beads with labels and metadata. Append-only sequenced events provide observation and replay. The default provider uses `bd` and Dolt, while file and executable providers keep the store boundary replaceable.

Yesod deliberately keeps three truths separate. PostgreSQL holds narrative intent, dispatch, runs, merge jobs, and telemetry. Beads/Dolt holds dependency structure. Git holds source ancestry. This costs more reconciliation code, but gives operationally distinct questions distinct records.

> **Design Trade-off — One substrate or several truths.** Gas City's universal Bead model simplifies composition. Yesod's split model makes the difference between intent, work readiness, execution evidence, and landed code harder to blur.

F.10. Cost and Model Selection

Gas Town supports multiple coding harnesses and static cost tiers. An operator can use cheaper models for patrol roles and stronger models for workers. Cost commands estimate retrospective session spend, but dynamic task-level routing remains more design than default operational policy.

Gas City separates harness, model, upstream provider, transport, and runtime into independent configuration axes. Current `main` records model tokens, compute time, and list-price estimates as usage facts and exposes `gc costs`. Model selection nevertheless remains primarily static per configured agent or pool. Cost observations do not yet close the loop into dispatch.

Yesod is more opinionated about the economic unit. A dispatch decision stores its lane and reason; the run ledger preserves every attempt; statistics correlate cost with merges and churn. The target metric is cost per accepted change, not price per token. Yesod's router

is still not a fully autonomous optimizer, but its data model is closer to learning which lane delivers value for a particular project and task class.

F.11. Knowledge Beyond Work Tracking

Gas Town and Gas City center orchestration. They are rich in tasks, formulas, sessions, mail, runtime state, and reusable configuration.

Yesod also treats software tools as a knowledge ecology. Tools have durable notes, feature requests, bug reports, processes, calls, topics, and cross-tool relationships. An agent can ask which tools exist, how they compose, what failed before, and which process should be poured. Tools can accumulate requests against one another.

Gas City's pack registry and Gas Town's federation ideas are adjacent, but they are not the same abstraction. A pack answers "how should this organization run?" The Yesod catalog answers "what tools and processes does this organization know, and what have they learned about one another?"

F.12. Authority and Trust

All three systems assume substantial local trust, but the boundaries differ.

Gas Town agents normally run as the user, share filesystem and network access, and can reach configured credentials and remotes. Role identity is conveyed through prompts, environment, directories, and Beads attribution rather than an unforgeable capability system. Its security documentation is explicit that stronger Polecat sandboxing remains future work.

Gas City improves the abstraction boundary and documents it carefully, but is intentionally not a sandbox. City configuration, imported packs, provider scripts, startup commands, and executable orders are trusted code. It strips ambient secret-looking environment variables from orchestrator-side helpers and offers more explicit provider and grant surfaces, but a privileged imported pack must still be reviewed like executable code.

Yesod's peer identity is also self-asserted, and role priming is not a credential. Its stronger property is a data-flow rule: peer mail carries no human authority, workers do not merge, and deterministic integration services own the normal automated path to `main`. That architectural separation reduces the number of agent sessions that require integration authority, although repository hosting must still enforce the desired credential policy if an absolute boundary is required.

F.13. What Yesod Should Borrow

Gas City contains several ideas that would strengthen Yesod without changing its identity:

1. **A runtime-provider interface.** Separate harness, model, upstream, transport, and execution location instead of encoding each lane as a largely bespoke command.

2. **Declarative packs.** Package role primers, processes, health policies, runtime presets, and tool-specific defaults as versioned imports.
3. **Controller-grade health patrol.** Add configuration fingerprints, crash-loop quarantine, provider-neutral liveness, bounded start/stop waves, and explicit recovery states.
4. **Formula control nodes.** Make fan-out, retry, drain, tally, and finalization first-class deterministic process operations.
5. **A sequenced event bus.** Preserve durable tables as truth while emitting a replayable, typed operational stream.
6. **Clear trusted-code documentation.** Treat imported workflow definitions and executable hooks as dependencies with explicit review and pinning rules.

F.14. What Yesod Should Keep

The comparison also clarifies what should not be generalized away:

1. **The deterministic AME.** Do not turn merge ownership back into an agent role that remembers Git procedures from a prompt.
2. **Independent proof of worker output.** Preserve commit, branch, acceptance, push, and ancestry checks outside the coding process.
3. **The run ledger.** Keep attempts, failures, tokens, cost, verification, and merge attribution as durable operational evidence.
4. **The catalog and process ecology.** Tool knowledge and reciprocal improvement are a different product advantage from orchestration packs.
5. **Delivered-outcome economics.** Optimize toward accepted changes rather than configured model price.
6. **The human-authority channel.** Keep authority separate from peer coordination and claimed role identity.

F.15. A Practical Integration Path

Gas City is most interesting to Yesod as a **session and workflow substrate**, not as a replacement for the factory's evidence core.

A future `yesod` pack could define the Mayor, planner, dispatcher, triage, infra-monitor, and experimental roles; provide their prompts and runtime defaults; and use Gas City pools and providers to keep those sessions alive. Version-2 formulas could express planning reviews, diagnostic swarms, documentation audits, and other workflows whose control structure benefits from deterministic fan-out and drain.

The pack would call Yesod's existing catalog and Beads-routing commands. Factory-dispatched code work would still enter the Yesod note/run ledger. The runner would still prove the branch. The AME would still own integration and deployment proof.

That composition preserves the best boundary in each design:

> **Gas City manages how agent sessions exist and cooperate; Yesod decides what constitutes proved software delivery.**

F.16. Sources

Official sources inspected for this appendix:

- Gas City README
- How Gas City Works
- Coming from Gas Town
- Gas City Health Patrol
- Gas City trust boundaries
- Gas City `gastown` pack
- Gastown pack Refinery prompt
- Gas Town README
- Gas Town architecture
- Gas Town Refinery manager
- Gas Town Refinery role
- Gas Town provider integration

G Glossary

Agent — an LLM session primed for a role and bounded by surrounding mechanisms.

AME — Autonomous Merge Engine: supervisor plus one-shot ephemeral merge workers.

Armed — a note whose `agent_dispatch` holds a real lane. Not a lifecycle status.

Bead — a work item in a per-tool Beads/Dolt graph.

Bead dependency — ordering between checklist items inside one feature's Bead tree; not a merge boundary.

Blocked — work stopped by a retry/no-progress/integration cap and needing changed conditions or judgment.

Catalog — PostgreSQL-backed registry of tools, notes, processes, runs, queue state, and telemetry.

Control plane — agents that make planning, routing, and diagnosis judgments.

Data task — a task whose acceptance probe, rather than Git commits, proves completion.

Dispatch — assigning work to an execution lane and allowing eligible runners to claim it.

Dispatcher agent — role that chooses and arms lanes; not the runner daemon.

Dolt — versioned SQL storage used by Beads workspaces.

Ephemeral merge worker — one-shot AME subprocess that performs one integration attempt.

ERS — judgmental ephemeral-refinery-supervisor agent; not the AME daemon.

Gate — must be qualified. The merge gate tests a candidate tree; Beads approval gates control work decisions.

Gate-on-merge — downstream work stays blocked until the prerequisite note lands or is superseded, even if its bead closed early.

Governing note — the feature note linked to a root Bead and armed for the checklist tree beneath it. The runner retains older exact-child compatibility, but planners no longer create child notes for ordinary checklist Beads.

Hold — explicit `agent_dispatch = hold`, removing a note from ordinary claims.

Lane — named model/backend target; not a host.

Merge job — durable record of one AME integration attempt.

Merge queue — durable but transient working set for branches awaiting integration.

Merge summary — durable record that a candidate landed.

Molecule — concrete Beads graph poured from a reusable process prototype.

Note — durable tool-attached knowledge or work record in PostgreSQL.

Note dependency — merge-aware ordering between independently mergeable feature notes; satisfied only when every prerequisite is `done` or `superseded`.

Process — reusable sequential or DAG composition of tool actions and handoffs.

Refinery profile — per-tool gate, repository, branch, deploy, and ownership configuration.

Runner daemon — `yesod codefactory-dispatch`; deterministic execution scheduler.

Run ledger — one durable row per dispatched execution attempt.

Service — deterministic long-running or one-shot mechanism.

Tool — catalog entity representing software the user owns, shares, or depends on.

Topic — durable cross-cutting knowledge object not owned by one tool.

Worker agent — coding process launched by a runner inside a worktree.

Worktree — isolated Git checkout for one execution or merge context.

H Building This Book

The book is deliberately reproducible with a small local toolchain:

- Pandoc 3.10 or compatible;
- XeLaTeX;
- Python 3 (standard library only) for deterministic SVG 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 deterministic SVG diagrams;
2. convert the diagrams to vector PDF figures;
3. run Pandoc with the custom KOMA-Script/XeLaTeX theme.

The exact Pandoc invocation and canonical source order are preserved in `book/Makefile`. The manuscript is split across `book/frontmatter/`, `book/chapters/`, and `book/appendices/`; `book/yesod-software-factory.md` is a human-readable source map rather than the manuscript body. Development waves live one-per-file under `book/appendices/development-waves/`. Metadata and geometry live in `book/metadata.yaml`; typography and the title page live in `book/tex/preamble.tex`; editable diagrams are generated by `book/figures/render.py`.

Useful validation:

```
make check
pdfinfo yesod-software-factory.pdf
pdftotext yesod-software-factory.pdf - | wc -w
```

H.1. Publishing an edition

The `Publish book edition` GitHub Actions workflow rebuilds and validates the book on macOS whenever canonical book sources change on `main`, and can also be started manually. It calls `scripts/publish-book.sh`, which:

1. chooses an unused immutable dated key in the `yesod-publications` R2 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 `yesod.work`.

If multiple 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; publication is not visible on `yesod.work` until that pull request is reviewed and merged.

For a portable edition, replace the macOS fonts with installed open fonts in `metadata.yaml`. The content does not depend on a proprietary template or network resource.