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 18, 2026

Revision 30acc5d

Stephen

Contents

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, nothing lands: the attempt terminates red, the note stays at `awaiting_merge`, and a retry becomes a fresh, independently-fenced attempt against whatever `origin/main` has since become—never a reopened one. Infrastructure failure is different again: if the remote gate cannot run, the gate attempt is not consumed and the work simply 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.**

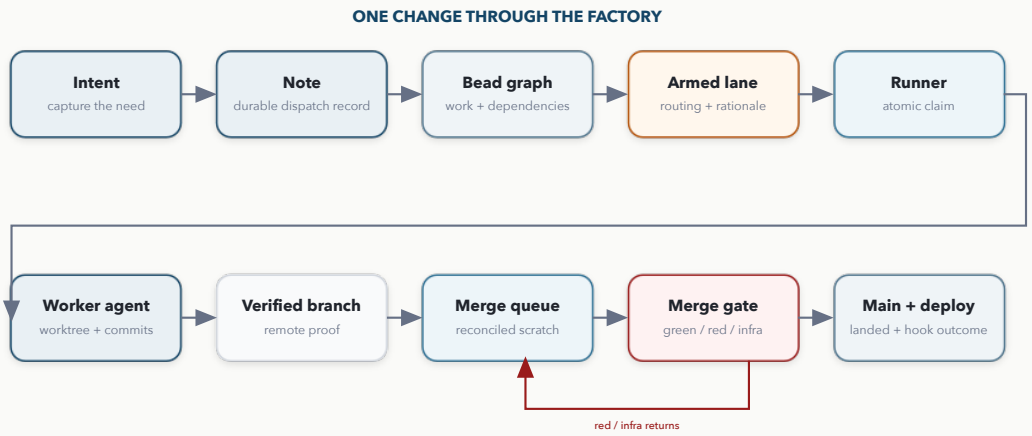


Figure 1.1.: One change moving through Yesod.

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

Stage	Evidence
Verification	acceptance/TUI result and remote branch proof
Integration	merge attempt, 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.

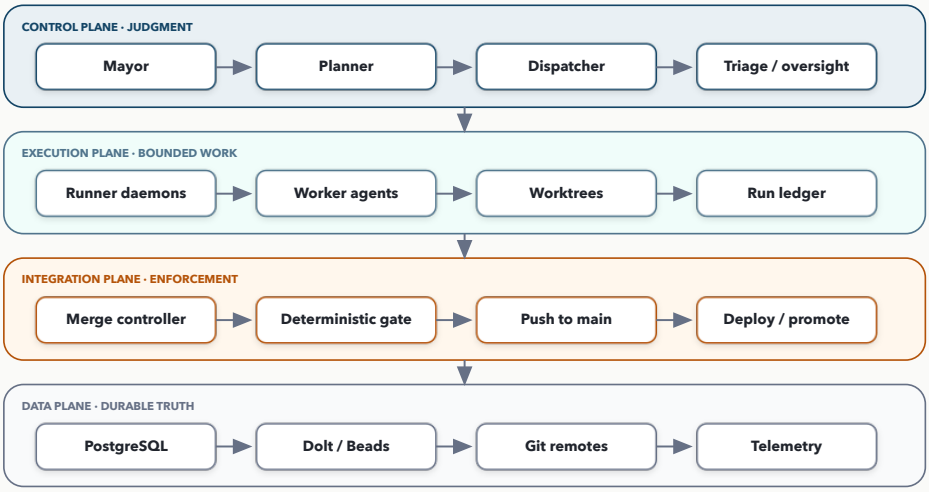


Figure 2.1.: The four planes of the factory.

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 nine 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
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 merge-controller 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 lanes and attempts, 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: Capturing Intent

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

A note is a PostgreSQL row with a stable, human-readable identifier such as `ys-yes-ab12`. That identifier follows the shape `ys-{project}-{hex}` and is composed of three parts:

- **`ys`** — the Yesod namespace, shared by every note in the registry;
- **`yes`** — the *project prefix*, a short code for the tool the note belongs to (`yes` is Yesod itself; another tool might use `sjb`, `clp`, or `fac`);
- **`ab12`** — a short hexadecimal code that makes the note unique within that project.

Read together, `ys-yes-ab12` says at a glance: "a Yesod note, for the `yes` project, with this unique id." 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 move through the **implementation lifecycle**; usage notes use a smaller knowledge lifecycle. A new feature or bug note is born `captured` — a raw intent, not yet planned. A planner takes it through `planning` and, once a validated plan is bound to it, `planned`; approving that plan for delivery promotes it to `open`. From `open`, a runner claims the note (`claimed`), executes it (`in_progress`), and hands the result toward integration — `awaiting_merge` when there are verified commits to gate, or `awaiting_verification`

when the outcome is ambiguous or intentionally has no commit. A green, gated candidate passes briefly through `merging` while the refinery finalizes it, and lands at `done`. `wontfix` and `superseded` are the remaining terminal states, for work that is abandoned or replaced.

Status is not the only axis a note moves along. A note also carries a **disposition** — normally `active`, but an operator can set it to `held`, `rework`, or `quarantine` to take the note out of ordinary play *without* moving it in the lifecycle. Status answers "how far has this work progressed?"; disposition answers "should it be running at all right now?" (Chapter 5 returns to holds.) Arming — assigning a lane — is a third, independent axis, covered below.

Two subtleties are worth flagging. A red gate does not move the note backward: it stays at `awaiting_merge` while the failure is recorded on the merge attempt, and a fresh attempt tries again. And a few status values — `blocked` is the clearest — are written by the database and runner in narrow situations without appearing in the canonical status list. This book therefore diagrams the transitions the factory actually drives, not a bare enumeration of allowed values.

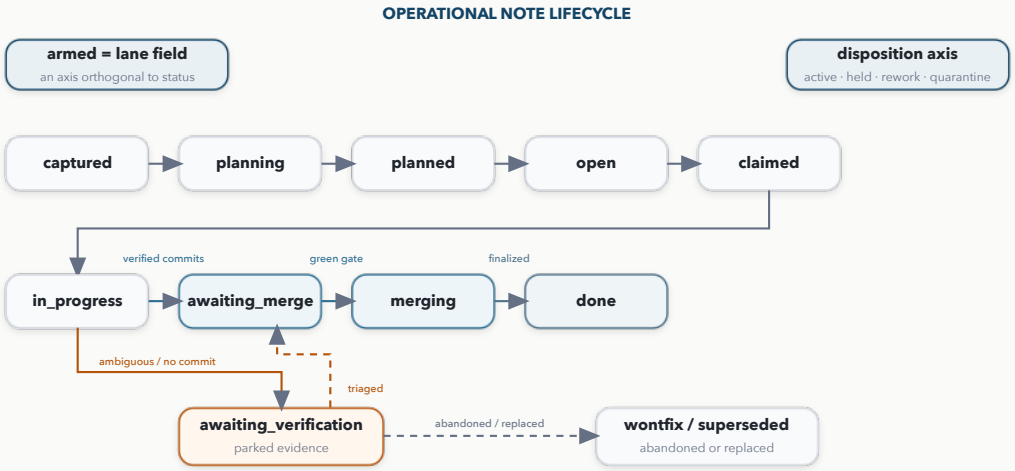


Figure 3.1.: The operational note lifecycle. Arming and disposition are shown separately because they are axes orthogonal to status.

> **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. "Hold" itself comes in two forms that are easy to confuse: the `agent_dispatch = 'hold'` sentinel, which suppresses one note on the dispatch axis, and the `held` *disposition* from the previous section, which an operator sets to park a note entirely. Both keep work from running, by different means (Chapter 5 uses the disposition form).

3.2. Beads: Planning the Work

If a note is *what* we want and *why*, a **bead tree** is *how* we intend to get there. Once an intent has been captured, a planner breaks it down into concrete, ordered steps and records that plan as a small tree of beads. This is the artifact a runner agent actually works against: the note points it at the feature to build, and the bead tree tells it what the pieces are and which must come first.

Beads is an open-source, dependency-aware issue tracker from Steve Yegge's Gas Town project, available at github.com/steveyegge/beads. Yesod embeds it as the planning substrate beneath every registered tool.

A bead is a task in a per-tool Beads workspace. Every mergeable feature note links to one **root bead**. The descendants of that root are the feature's internal checklist: source changes, tests, documentation, or other concrete steps that should land together. Dolt — a Git-style versioned SQL database — stores those workspaces, so a plan carries history the same way code does.

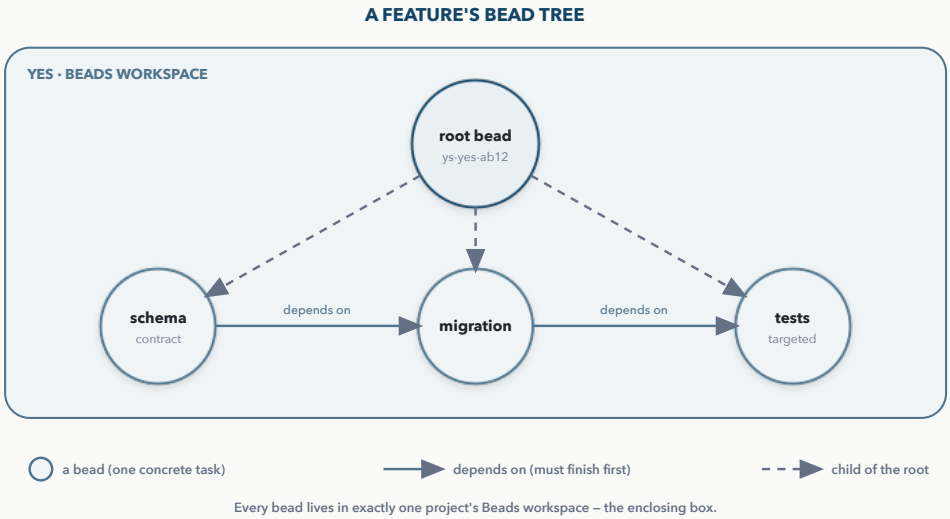


Figure 3.2.: A feature's bead tree. Circular nodes are beads; directed edges are dependencies that order the work; every tree belongs to one project's Beads workspace.

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.

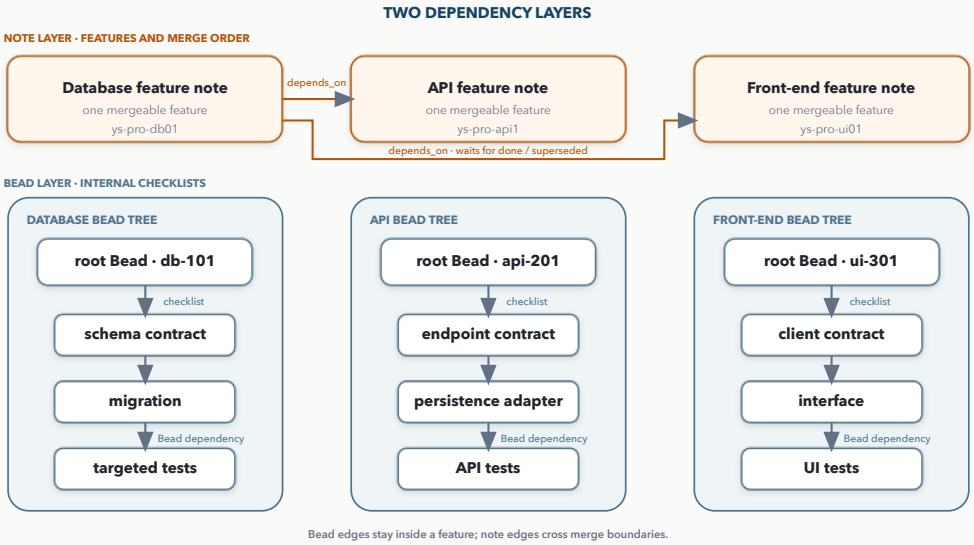


Figure 3.3.: Note dependencies order mergeable features; Bead dependencies order checklist work inside each feature.

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 a set of routing constraints, each of which exists for a concrete reason:

- **explicit host pins** — so a task that needs one machine's local database, checkout, or secret runs only there;
- **soft host affinity that ages out** — so follow-up work tends to reuse the host that already has a warm worktree, without being bound to it forever;
- **per-model host plans** — so a model that is only licensed or configured on certain hosts is dispatched only to those;
- **tool workspace availability** — so a note is never claimed on a host that has not checked out that tool's repository;
- **note-level dependencies** — so a feature that depends on another does not start until its prerequisite has actually landed;
- **retry backoff** — so a task that just failed is not immediately re-claimed in a tight, wasteful loop;
- **the `hold` sentinel** — so an operator can deliberately keep a note out of dispatch while it is being reviewed.

Keeping lanes separate from machines also solved an early identity problem. Each runner has its own stable ID, and dispatch records that ID so the run ledger knows *which machine* actually did the work. Early on it was tempting to let one host borrow another host's runner ID, so that a task pinned to the second host would get picked up by the first. That is the wrong fix: it makes the ledger lie about where work ran. The place to say "this task prefers that host" is the routing data — host pins and affinity — never the host's identity. A host advertises the lanes it can serve; it never impersonates another runner to attract a task.

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 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. Each attempt is a fresh, fenced record; authority lives in the attempt rows and their per-lane leases. Work enters through an intake queue, but that queue is only a waiting room — the controller turns each admitted item into an attempt, and it is the attempt that carries the truth.

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, merge attempts 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**.

The Yesod software factory is impressive — but the older, simpler use of Yesod as a *tool registry* is just as important. That layer is easy to lose sight of once runners and MicroVMs enter the story, yet it is what makes Yesod more than a job queue with an LLM attached.

4.1. Tools as Durable Context

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

A naïve agent gives up here, or papers over the failure and ships a broken post. But this agent knows something important: `scr-wrap` is a Yesod-registered tool — part of the very factory the agent is working in. The failure is therefore not a dead end but something the agent can *act on*. It files a bug report against the offending tool:

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

Yesod records the report and hands back a **note ID** — say `ys-scr-4f2a`. That ID is a ticket: it is addressable, it has a status, and, in the direction this registry is built toward, it comes back with an estimate of when the fix should land. The agent does not have to hold the problem in its context or block forever. It keeps the ticket, sets the screenshot subtask aside, and gets on with the parts of the post that do not depend on it.

Meanwhile the ticket flows through exactly the machinery the rest of this book describes: the bug report becomes a bead, a runner claims it, the fix is gated and merged. Every so often the waiting agent simply asks Yesod how its ticket is doing:

```
yesod notes show ys-scr-4f2a --json   # → "status": "done"
```

When the status reaches `done`, the agent re-runs `scr-wrap`, which now accepts the wide screenshot, and the post is finished. No human relayed the bug between the two tools. No fix was

lost in a transcript. One tool's shortcoming became a tracked, closed piece of work — and the agent that hit it was able to keep going. That loop, from friction to filed ticket to observed fix, is the vision the tool registry makes possible; the rest of this chapter describes the durable substrate underneath it.

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.

The blog-post scene is not a special case. Suppose instead 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. On the roadmap, this retrieval is slated to grow richer still: a multi-vector embedding setup would match a question against notes and process steps along several semantic dimensions at once, sharpening which evidence the model is handed before it reasons.

> **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.

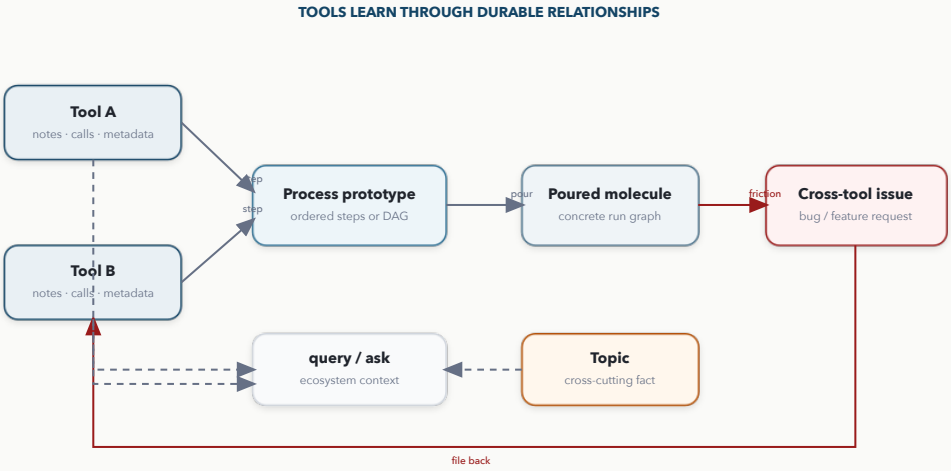


Figure 4.1.: Tools, processes, concrete runs, and cross-tool feedback form a learning loop.

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: the judgment of reading source and shaping work is theirs, while the orchestration around them is mechanical. When a note needs a plan, a durable **planning job** is opened, and a standing `codefactory-dispatch` runner's planning controller claims that job under a lease and spawns a planner agent—by default a Codex session[1]—to ground the request and return a proposed work graph. That controller runs on the `yesod-runner-*` coding fleet, not on the operator-only `yesod-dispatch` host.[2]

Planning enters two ways. Most work flows through the durable subsystem above: the mayor, or an automatic policy at note creation, requests planning on a note[3], and the controller

[1] The default planner target is `codex-gpt-5.6-sol-xhigh` (`yesod/src/yesod/planning_execution.py`, `DEFAULT_PLANNER_TARGET`). The durable provider requires a `codex…` target and launches the planner as a local subprocess (`LocalCodexPlanningProvider`), not a long-lived `opencode` session.

[2] The planning controller is a tick inside a standing `codefactory-dispatch` runner whose config sets `planner_enabled: true` (`yesod/src/yesod/runner.py:992`; the tick is `_tick_planning`, invoked from the main loop at `runner.py:1128`). It is deliberately *not* operator-gated — unlike the lifecycle scheduler (`runner.py:1013`, `if config.lifecycle_scheduler_enabled and not config.operator`) — but the operator VM `yesod-dispatch` sets `planner_enabled: false`, so planning executes on the `yesod-runner-*` coding fleet, alongside coding work. Plan/schedule *authority* is still anchored on `yesod-dispatch` via the lifecycle-v2 enable token (`reference/live-topology.md`).

[3] `request_planning` opens the durable job and moves the note `captured → planning` (`yesod/src/yesod/note_lifecycle_ops.py:264`, calling `planning_jobs.ensure_job`). It is invoked by `yesod tool <tool> note-plan` (`yesod/src/yesod/tool.py:1487`) and automatically when a feature-request or bug-report note is created (`tool.py:423`).

spawns a fresh planner agent for that job. For work that benefits from explicitly chosen context, expertise, or attention, an operator can instead drive an **interactive planner**—an agent primed with the planner role that grounds the note and coordinates with the mayor through `yesod mail`.[4] Either way the planner reads the target repository and turns durable intent into a work graph; the entry 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.

The note sits in `planning` while the planner grounds the feature. When the source-grounded plan, scope fences, acceptance checks, and Bead descriptions are complete and validated, the note becomes `planned`; approving that plan for delivery promotes it to `open`, still unarmed. Lane choice and arming belong to the dispatcher.

These transitions are more than a planner editing a status field. The controller validates the returned plan against the note's current revision before the note becomes `planned`, and records every step on the durable job.[5] The job, not the conversation, is the source of truth: if a planner agent dies mid-plan, the planning job's recorded state and validated artifact survive, and the controller can retry or supersede the attempt without losing work.

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:

[4] The interactive planner role is primed with `yesod prime --planner`; it grounds work and signals the mayor by mail (`yesod/src/yesod/prime.py`). This is the human-in-the-loop path, distinct from the default durable Codex path.

[5] Planning jobs carry their own lifecycle — `queued → running → succeeded / failed / superseded / cancelled` — under leases and fencing (`yesod/src/yesod/planning_jobs.py`). The `planning → planned` transition is performed by the `planning_validator`, and `planned → open` by the operator or `auto_promoter` (`yesod/src/yesod/note_lifecycle.py`).

- 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 ys-yes-ab12 --target LANE \
  --reason "advanced lane: query crosses note, Beads, and runner state"
```

The reason matters as much as the target. A sentence like the one above 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.[1] 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;

[1] The dead-runner threshold is `DEFAULT_DEAD_THRESHOLD = 300` seconds (`yesod/src/yesod/runner.py:85`), but recovery cross-checks the run ledger so a busy-but-live worker is not reaped; `running` rows left by a genuinely dead daemon are settled to `lost` by `reconcile_lost_runs` (`yesod/src/yesod/run_ledger.py:677`).

- 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[2] — a deadlock is a transient lock-ordering conflict that simply succeeds on a retry, unlike a logic failure, which must not be retried blindly.

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.

[2] The claim runs inside a `_claim_tx` wrapped by `_with_deadlock_retry` (`yesod/src/yesod/runner.py:2033,2574`); the row lock is a `SELECT … FOR UPDATE` on the matching `tool_notes` row (`runner.py:3598`). Only serialization/deadlock errors are retried.

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.[3] 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, model and agent[4], timestamps, status, exit code, worktree, result commit, token and cost data, verification results, outcome verdict, failure classification, and log archive details.

[3] The defaults live in `yesod/src/yesod/runner.py` — e.g. dead-runner threshold `DEFAULT_DEAD_THRESHOLD = 300` (`:85`), worker runtime cap `DEFAULT_TASK_TIMEOUT = 1800` (`:86`), SIGTERM-then-SIGKILL grace `DEFAULT_TERM_GRACE_SECONDS = 30` (`:87`), max execution retries `DEFAULT_MAX_RETRIES = 3` (`:96`), soft-affinity age-out `DEFAULT_SOFT_AFFINITY_TIMEOUT = 300` (`:107`), and the stalled-note reaper `DEFAULT_REAPER_THRESHOLD = 300` (`:110`).

[4] The `runs` table has `model` and `agent` columns but no dedicated `lane` column; the selected lane is recorded in the `agent` column (`yesod/src/yesod/db.py`, `runs` schema).

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.[1]

When a limit is crossed, the runner sends SIGTERM and, after the configured 30-second grace period, SIGKILL to the process tree.[2] 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.[3] Those are two durable operations rather than one

[1] The governance loop watches the wall-clock deadline (`yesod/src/yesod/runner.py:4715–4721`), durable cross-host kill requests (`:4740–4759`), the token budget (`:4765–4778`), and process exit (`:4723–4724`).

[2] The grace period is not a watched condition but the SIGTERM-then-SIGKILL window applied by `_terminate_process_tree` (`runner.py:4780–4800`); its default is `DEFAULT_TERM_GRACE_SECONDS = 30` (`runner.py:87`).

[3] `merge_queue` is the intake table; the deterministic controller is now its only consumer, bridging each enrolled item into a fenced merge attempt (`yesod/src/yesod/refinery_intake.py`). Enrollment (`runner.py:7454–7473`) is a separate durable operation from the `awaiting_merge` status write (`:7451`), which is why reconciliation may need to rebuild it.

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 transient events such as rate-limit backoff and claim collisions can avoid consuming the ordinary allowance.[4] 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:

[4] The retry-sparing paths are specifically rate-limit backoff (`_handle_rate_limit`, `runner.py:7601–7660`) and claim collisions (`_handle_claim_collision`, `:5775–5789`). A generic infrastructure *verdict* still increments the counter in `_handle_failure` (`:7769`). The cap is `DEFAULT_MAX_RETRIES = 3` (`:86`); backoff grows as `2**retries` minutes (`:7794–7805`).

> **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 that simple, for three reasons the factory hits constantly.

- **A false positive 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 must be 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

By now the diagnostic-command note has travelled the whole pipeline: it was captured, planned into a bead tree, armed, and dispatched to a coding agent, which worked in an isolated worktree and pushed a branch it believes completes the task. The runner verified that branch and advanced the note to `awaiting_merge`; the controller then claimed the lane, merged the branch onto current `main`, and built one exact candidate tree. That candidate — a specific, fetchable SHA, not "the branch" — is now ready to be judged, and 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 own repository the provider is the MicroVM backend — the high-fidelity path that trades speed for a clean, isolated world. 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

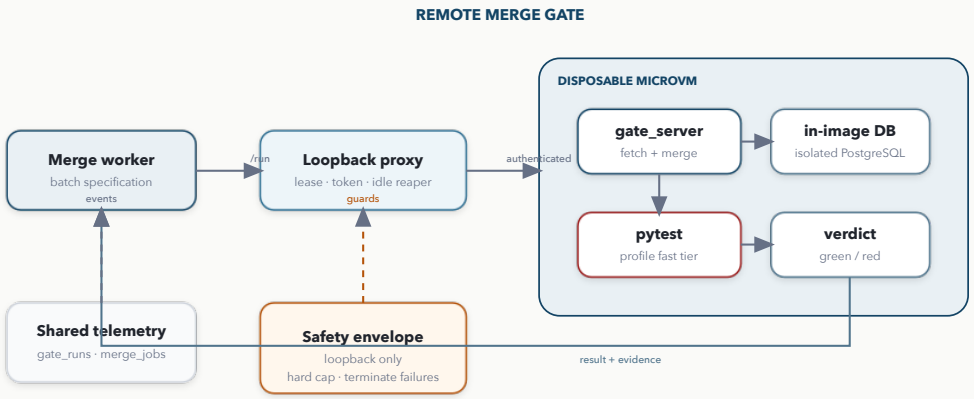


Figure 9.1.: The MicroVM merge-gate path and its safety envelope.

It is worth pausing on *why* a high-fidelity gate pays for a whole virtual machine per run. 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

Both backstops — the cap and the tripwire — rest on one humble question: *how many MicroVMs are running right now?* If that number is wrong, both fail at once. And for a couple of days it was wrong in the worst possible direction. The function that counted the fleet looked for the instance list under one field name; the cloud API had started returning it under a different one. The two had quietly drifted apart, so the counter would read *zero* while real machines were live and burning money — and a cap that believes the fleet is empty never fires.

That bug is fixed today: both sides read the same field, and a regression test now locks them together so they cannot drift apart again. The lasting lesson is simpler than the bug. A safety mechanism is only as good as the *live* API contract it reads — a mock that returns the shape you expected proves nothing about the shape the cloud actually sends. What made the fix stick was turning that contract into 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 Landed Is Not Delivered

"Deploy" is a slippery word. For many projects it means something small and local — copy a new executable over the old one, restart a service, or simply let a GitHub Action run to green and call the release done. Roll one of those back and you have undone the single thing you changed. But when the system being deployed is the *factory itself* — Yesod updating its own live fleet with the code it just merged — deploy becomes a coordinated move across several hosts and a shared database, where failure is partial rather than total. This chapter is about that harder case.

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. You might reach for an off-the-shelf blue-green or rolling deploy here — but those assume the system being deployed is separate from the one doing the deploying. Yesod is promoting its own control plane onto its own hosts against a shared database: the deployer is inside the blast radius, which is exactly what the transaction is built to survive. 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

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

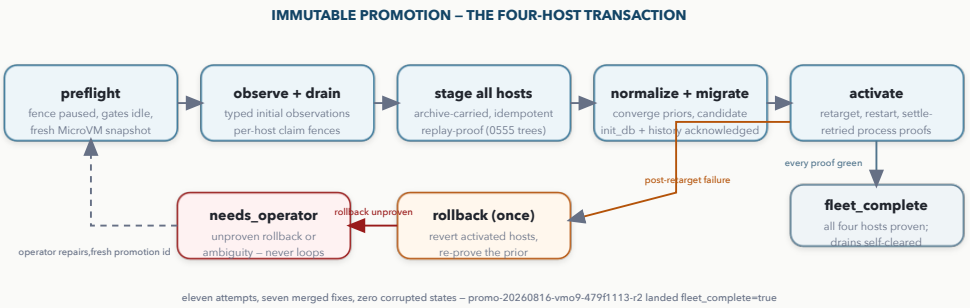


Figure 10.1.: The four-host promotion transaction.

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.

This is not a thought experiment. The promotion machinery's first real end-to-end execution was an **eleven-attempt campaign** in which every stop was a contract being discovered, filed, and fixed — and across all eleven, not one attempt left a corrupted or half-moved fleet. Chapter 17 tells that campaign in full; what matters here is that the design's central claim — a failed proof only ever *stops* — was earned under exactly the partial failures this chapter describes.

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.

> **Factory Floor — Host placement is part of correctness.** An earlier watchdog checked worker liveness by calling `os.kill(pid, 0)` on a PID read from a shared table — which silently assumes every process lives on the host doing the checking. Across a real fleet that assumption is false: a watchdog can "confirm" a process that is actually running on another machine, or reap the wrong one. Consolidating the control plane onto a single Conductor removed the cross-host guessing, but the durable lesson outlives that fix — *where* a process runs is a correctness fact, not a deployment detail, and any cross-host operation has to carry host identity, not infer it.

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

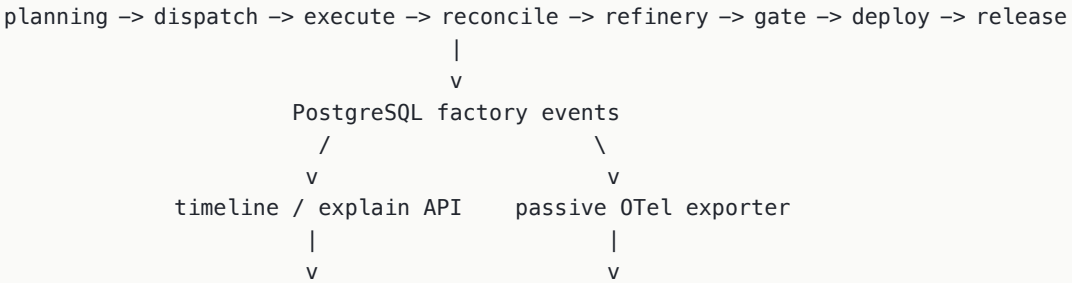
Yesod is observable through several surfaces because no single table contains the whole truth. The next step is not to invent one perfect dashboard. It is to give every important transition a common, causal observation record while preserving the authority of the systems that already own the facts.

> **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

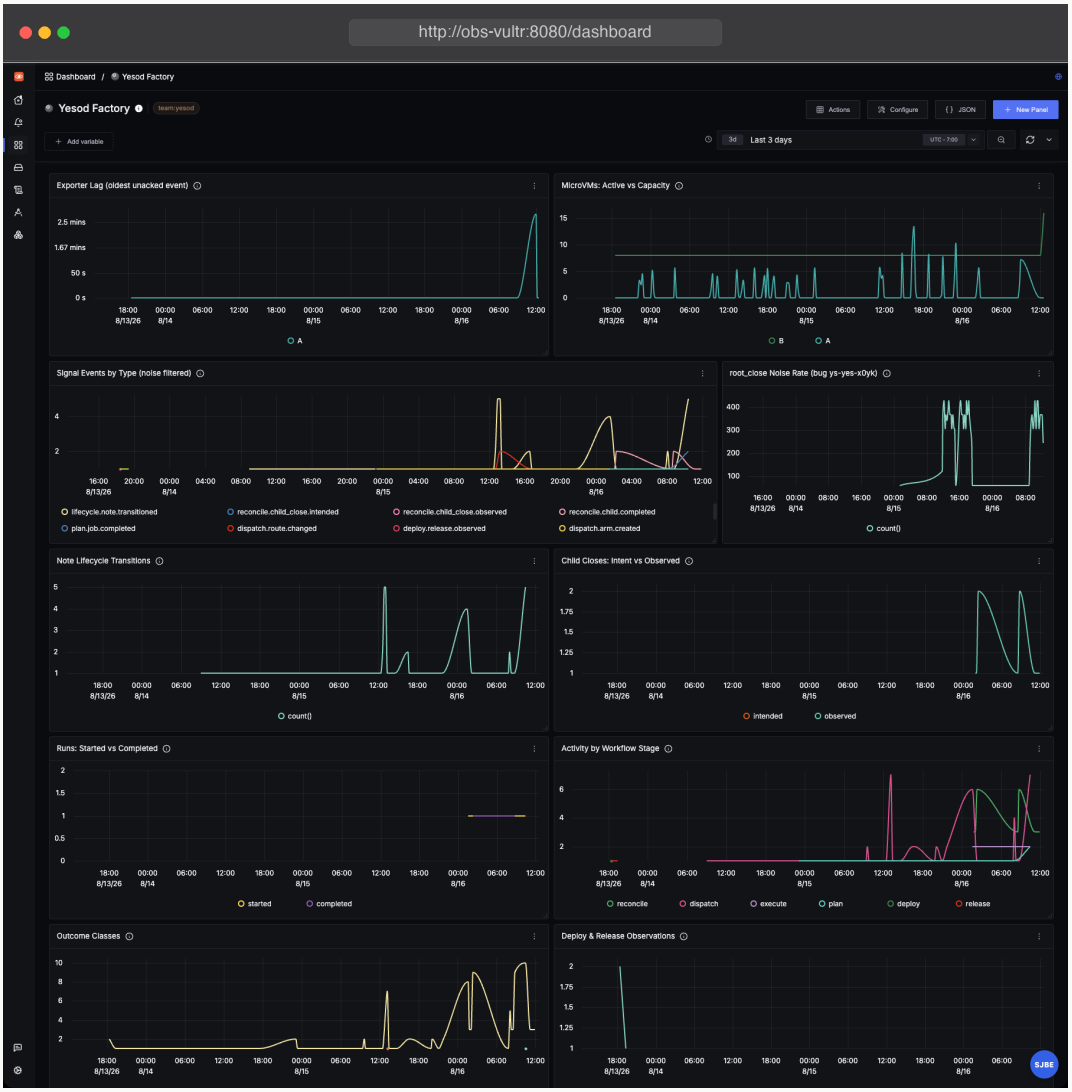


Figure 11.1.: The Yesod Factory dashboard over its first three days of production.

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.

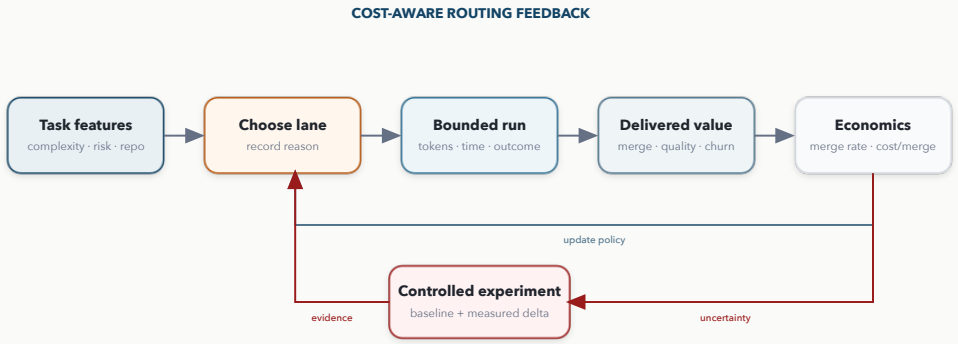


Figure 13.1.: The cost-aware routing feedback loop.

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

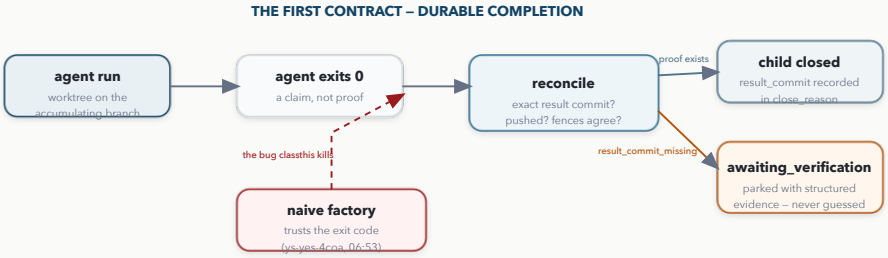


Figure 15.1.: The durable-completion contract.

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.

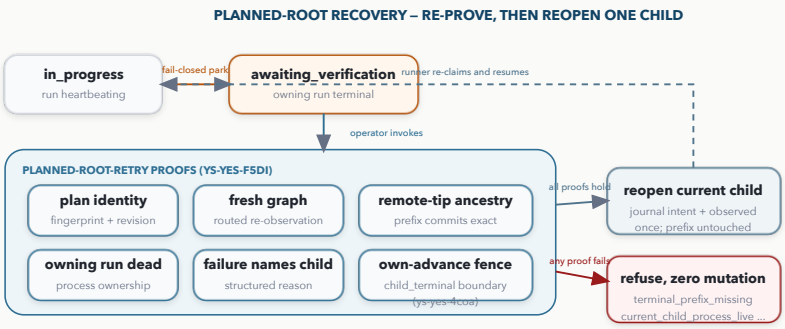


Figure 15.2.: Planned-root recovery.

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

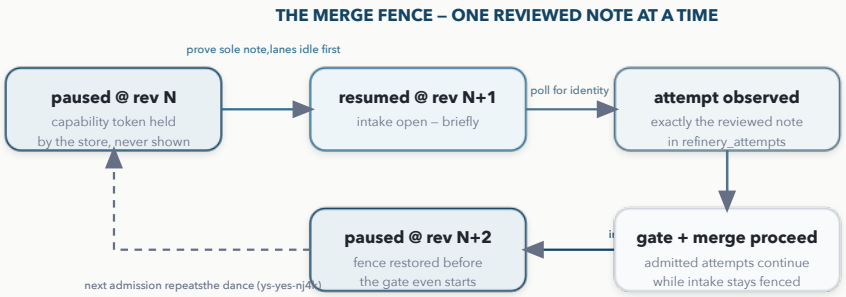


Figure 15.3.: The merge fence and the promotion transaction.

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.

Yesod has several purposes. As a **factory**, it is a way to make the best possible use of both near-frontier and economy models — to make software development both efficient and controlled. It is also, separately, interesting as a **registry of tools**: the near future it points toward is one where you hand an agent an objective and it works out, in a disciplined way, which parts of that objective existing tools already accomplish and which new ones need to be invented. The most interesting future, then, is not "more agents" but 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.

The *holy grail* here is accurate **complexity estimation**. Like many problems in computer science and mathematics, it sounds easy and is anything but — yet Yesod does not need a perfect answer, only a *good enough* one. That is the deeper role: Yesod is a platform for experimenting with how to use models of very different costs, and even a rough, improving estimate of complexity is enough to route most work to the cheapest model that can still do it well.

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.

The planned R740xd server points directly at this: with a large local pool of compute, agents could stand up *simulated* factories and experiment against them freely — testing policies at a scale, and with a safety, the live fleet can never offer.

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. The observability initiative already under way is the nervous system this depends on — the signals it collects are precisely what let a cause fingerprint be computed in the first place.

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. Much of this is closer than it sounds: most of the pieces — note-and-bead lineage, route and model identity, commits and tests, gate image and recipe digests, merged SHA, deploy artifact — are already captured today; what remains is binding them into a single signed attestation.

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: most of what earlier editions listed here has already shipped under the reliability redesign — the standing AME — the Autonomous Merge Engine, the always-on supervisor-plus-worker merge daemon of earlier editions — gave way to the deterministic controller, the gate proxy became a managed service, deploy-and-runtime proof arrived as the four-host promotion transaction, and the MicroVM fleet count is now snapshotted with its counting contract pinned by a regression test. What is genuinely left near-term is narrower: make any residual watchdog action host-addressed, wire up retention cleanup, close the image-provenance gap, and add provider redundancy so no single credit window can stall the pipeline.

A single tension runs through all of these: *let an agent do it* versus *make the behavior deterministic code*, even when that code grows large. Yesod deliberately errs toward the large deterministic system — with agents reserved for the genuinely undecidable seams — because a determinism you can test and reproduce is worth more than a cleverness you have to trust.

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. Further out still: let an agent *request a new feature from the factory itself* and have it planned, built, gated, and delivered in a timely, cost-effective way — the factory extending its own capabilities on demand — and, more broadly, let the factory manage the disciplined creation of *any* AI-creatable output, not only code.

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.

And it keeps faith with the name: *yesod* is Hebrew for **foundation** — which is precisely the ambition, a foundation for AI tools and for the work of building with them.

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
`captured`	a new feature/bug note, not yet planned
`planning`	a planner is grounding it into a bead tree
`planned`	a validated plan is bound; awaiting approval
`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`	a verified branch is queued for the gate
`merging`	the gate is green and the controller is finalizing the merge
`done`	landed, synchronized to Beads
`blocked`	stopped by a retry/no-progress cap; needs changed conditions or judgment
`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

Value	Meaning
`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 Attempt State

Integration authority is the **immutable merge attempt**, not the intake queue. Each attempt steps through:

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

with terminal states `gate_red`, `push_raced`, `needs_rework`, and `infrastructure_failed`. Terminal attempts never reopen; a retry is a new, freshly-fenced attempt. The intake `merge_queue` (`queued`, `gating`, `gate_red`, `held`) is a waiting room the controller bridges into attempts — it is not the authority.

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

Mechanism	Default / observed contract
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
Merge controller tick	30 s
Queue gate-red retry cap	2
Stale blocked inspection floor	10 min
Retry-capped reopen window	120 min
Remote gate attempts	2
Gate proxy idle reaper	300 s
Gate hard fleet cap	3 configured default; integration must be verified
SPS loop	60 s

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	the merge controller
refinery	human-guided merge orchestration	the deterministic merge controller
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
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

Service	Mechanical responsibility
merge controller	claim a lane under lease, step each immutable attempt (prepare, gate, push), reconcile per tick
promotion worker	stage all hosts, migrate, activate host-by-host with proofs, or roll back
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 merge controller 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`
Merge controller	`yesod/src/yesod/refinery_controller.py`, `refinery_scheduler.py`
Merge attempts and gate broker	`yesod/src/yesod/refinery_state.py`, `refinery_gate_broker.py`

Claim	Source anchor
Promotion transaction	`yesod/src/yesod/refinery_deployment.py`
DB-backed profiles	`yesod/src/yesod/refinery_profiles.py`
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 Glossary

Agent — an LLM session primed for a role and bounded by surrounding mechanisms.

AME — *Autonomous Merge Engine* (retired): the earlier always-on supervisor-plus-ephemeral-worker merge daemon, replaced by the deterministic **merge controller**.

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 — *(retired)* the AME's one-shot subprocess that performed one integration attempt; superseded by the controller's immutable **merge attempts**.

ERS — *(retired)* the ephemeral-refinery-supervisor agent that gave judgmental oversight of the AME; deterministic recovery is now the **operator inbox**.

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 attempt — an immutable, fenced record of one integration try under the merge controller; terminal attempts never reopen, and a retry is a new attempt.

Merge controller — the deterministic, PostgreSQL-authoritative daemon (`yesod-refinery-controller`) that gates a candidate and lands it under a per-lane lease; successor to the retired AME.

Merge queue — durable but transient intake the controller bridges into fenced attempts; a waiting room, not the authority.

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`.

Operator inbox — the deterministic recovery surface: a bounded queue of decisions the controller opens when automated recovery would have to guess, rather than an agent improvising against production.

Process — reusable sequential or DAG composition of tool actions and handoffs.

Promotion — the single durable, restart-safe transaction that moves the whole host fleet from one immutable release to another, or rolls back; how Yesod deploys itself. Distinct from merge.

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.

F 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. The development-wave histories are kept as reference under `book/appendices/archive/developmen` `waves/` but are no longer part of the rendered manuscript. 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
```

F.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.