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 · July 2026

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 engine 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	`cf091d2e68488d640e59e3d24b31b98354d08812`
Planning-contract update	`a9f19543d6dba7d880a2d6c35cbd559597e248ae` (`skill-prime-yu4c`)
Re-verification date	2026-07-15 PDT
Deployment observation	2026-07-11 PDT / 2026-07-12 UTC
Documentation project	`yesod-aicoe-docs`

The requested development commit, `a7870720`, and the baseline commit above have identical trees. The later identifier records the merge of the MicroVM launch-storm hardening onto `main`. The planning, dependency, compaction, and handoff sections were subsequently re-verified against `a9f19543`, the focused Yesod feature commit that makes one note the independently mergeable feature unit and its Bead tree the internal checklist.

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.

This documentation update did not deploy or operate production services. Its revised planning doctrine reflects the committed feature branch named above; deployment observations

remain dated separately. 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 Autonomous Merge Engine—**AME**, the deterministic core of the refinery—takes over. A small supervisor reconciles per-tool queues and launches a one-shot ephemeral merge worker. The worker resets its checkout to current `origin/main`, selects a compatible batch, verifies each branch, merges the candidate commits locally, and runs the repository profile's gate.

On green, it pushes the merged tree to `origin/main`, verifies the push, and verifies that each note's commits actually landed. It writes durable merge evidence, advances notes to `done`, and runs the profile's post-merge hook when configured. A deploy hook can report `deployed`, `skipped`, or `failed`. A failed hook does not rewrite Git history; it is recorded and escalated because the merge has already happened.

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

The whole path can be compressed into one sentence:

> **Intent becomes a note and a bead graph; arming makes eligible work visible to runners; a runner produces a verified candidate branch; the AME gates and lands it; a post-merge hook records the deployment outcome.**

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:

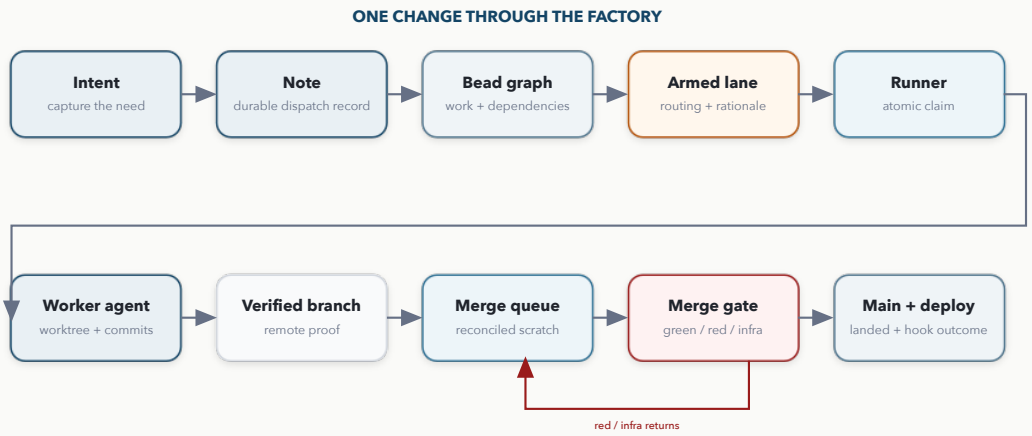

Figure 1.1.: One change moving through Yesod.

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

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

1.2. The Proof Chain

Each stage produces evidence for the next:

Stage	Evidence
Planning	note–bead linkage and dependency graph
Dispatch	audited lane assignment and reason
Claim	locked note row, runner ownership, bead assignment
Execution	run ledger row, log, progress events, commits
Verification	acceptance/TUI result and remote branch proof
Integration	merge job, gate run, pushed SHA, landed-commit check
Post-merge action	structured hook outcome and log

The last row proves what the hook reported, not necessarily that users can exercise the code; Chapter 10 returns to that distinction. More broadly, this chain is the difference between an

agent demo and a factory. A demo can end when the model prints a confident summary. In a factory, every handoff must carry durable evidence that the receiving stage can verify.

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

2 Judgment and Mechanism

Yesod divides the system into four conceptual planes.

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

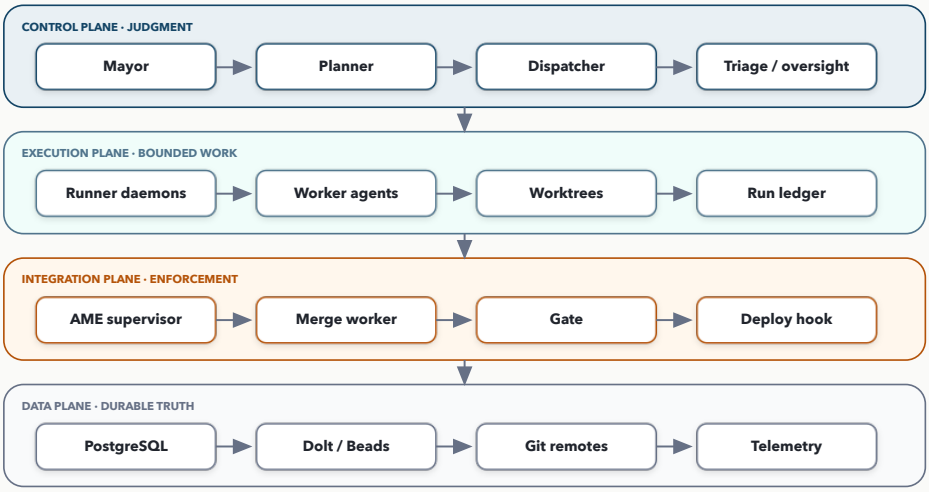

Figure 2.1.: The four planes of the factory.

2.1. The Control Plane

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

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

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

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

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

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

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

2.1.1. Bootstrapping a role with `yesod prime`

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

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

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

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

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

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

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

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

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

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

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

That rule prevents a compromised or confused agent from turning a plausible message into a production command. Deterministic services such as the AME 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 AME, the merge gate, and the post-merge hook.

The supervisor is intentionally small. It reconciles durable state, selects a tool to drain, respects the global merge lock, and spawns an ephemeral merge worker. That worker starts

fresh and loads current code for every attempt, builds a batch from source-of-truth state, performs the merge and gate, records the result, and exits.

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

2.4. The Data Plane

Three stores cooperate without pretending to be one database:

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

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

2.5. Why Determinism Matters

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

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

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

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

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

3 The Two Currencies

Yesod coordinates work with two first-class currencies: **notes** and **beads**.

Notes carry intent, lifecycle, and dispatch history; beads carry work structure and dependency readiness. They are linked, but they are not interchangeable.

3.1. Notes: Intent and Dispatch History

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

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

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

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

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

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

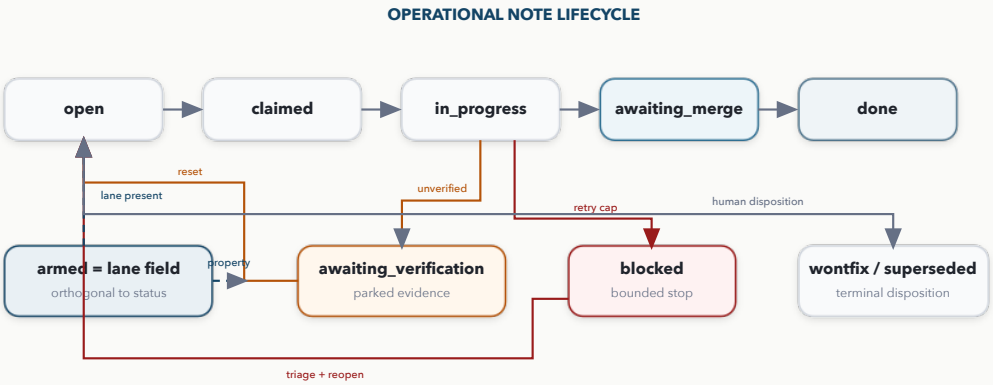

Figure 3.1.: The operational note lifecycle. Arming is shown separately because it is not a status.

3.2. Beads: Work and Dependency Structure

A bead is a task in a per-tool Beads workspace. Every mergeable feature note links to one root Bead. The descendants of that root are the feature's internal checklist: source changes, tests, documentation, or other concrete steps that should land together. Dolt provides a versioned SQL backend for those workspaces.

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

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

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

3.3. Two Dependency Layers

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

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

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

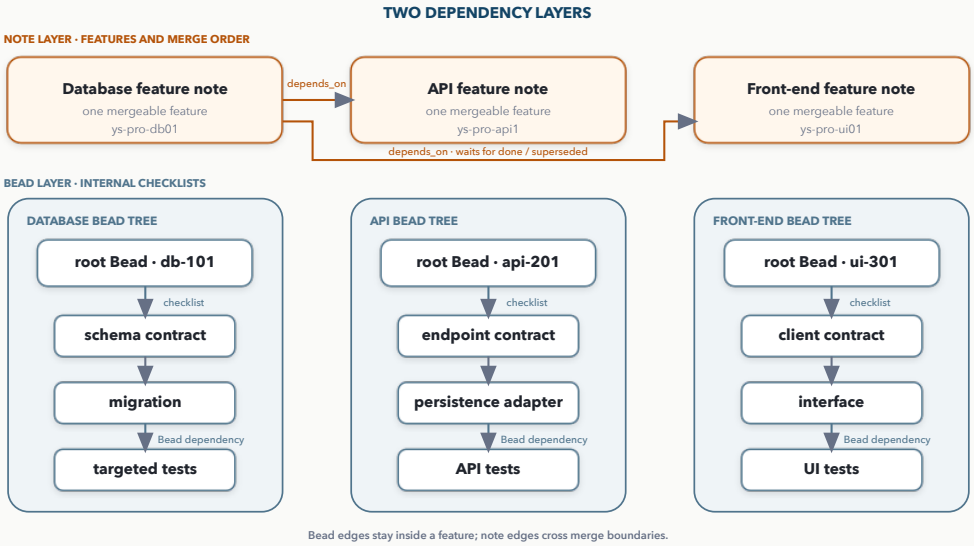

Figure 3.2.: Note dependencies order mergeable features; Bead dependencies order checklist work inside each feature.

The boundary rule is simple: if a planned child must start from another change already merged to `main`, must be routed independently, or deserves its own merge decision, it is not merely a child. Promote it to a separate feature note with its own root Bead, then connect the feature notes with `depends_on`. Ordinary checklist Beads do not get child notes.

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

3.4. Lanes, Hosts, and Affinity

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

The runner also respects:

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

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

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

3.5. Three State Machines, Not One

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

1. **Note lifecycle** — the active path through `open`, `claimed`, `in_progress`, `awaiting_verification`, `awaiting_merge`, and `done`, plus side dispositions.
2. **Bead graph** — open or closed, ready or blocked by dependencies.
3. **Merge queue** — operational states such as `queued`, `gating`, `gate_red`, `blocked`, and `held`, or no row at all.

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

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

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

4 The Registry Beneath the Factory

The software factory is built on an older and more general idea: **a personal registry in which tools can be described, questioned, composed, observed, and improved together**.

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

4.1. Tools as Durable Context

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

The basic operations are intentionally ordinary:

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

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

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

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

4.2. Querying One Tool in Its Ecosystem

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

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

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

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

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

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

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

4.3. Processes as Molecules

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

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

Without explicit dependencies, the steps form a sequential chain. A declaration such as `--dep 3:1,2` says that step 3 depends on steps 1 and 2 and switches the process to DAG authoring. Independent steps become parallel roots or branches; later steps can depend on one or several predecessors. A step can also be a human handoff rather than a tool action.

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

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

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

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

The lifecycle has a small command vocabulary:

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

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

4.4. Composition Creates a Feedback Network

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

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

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

Runs teach the catalog about behavior. `yesod call TOOL ...` logs an invocation with command, result, timing, session, and bounded output tails. Telemetry can mine repeated failures into draft bug reports or process candidates.

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

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

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

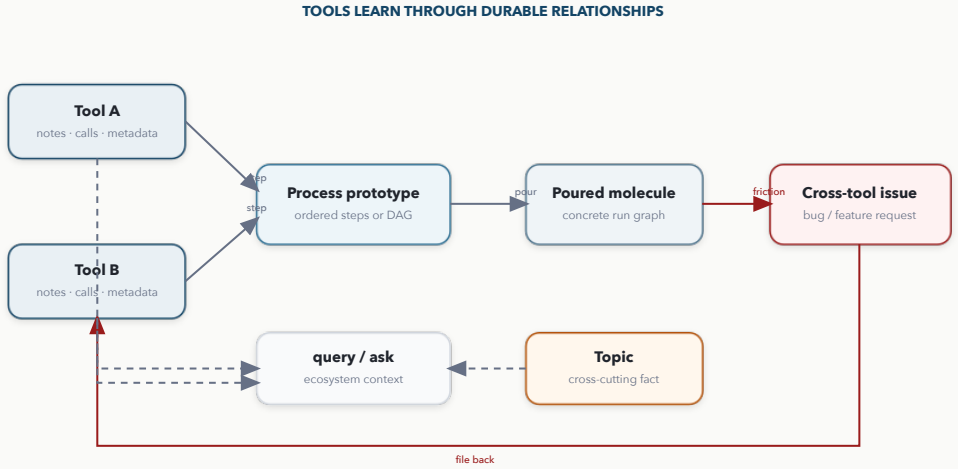

Figure 4.1.: Tools, processes, concrete runs, and cross-tool feedback form a learning loop.

4.5. Topics: Knowledge Without a Tool Owner

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

The ownership rule is simple:

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

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

4.6. More Than a Factory Front End

The original registry remains useful even if no coding agent is running. It can answer how a local ecosystem works, preserve operational memory, compare tools, expose unresolved needs, and compose repeatable processes. The software factory is one consumer of that substrate: the consumer that turns selected notes into code.

> **Invariant — The catalog is not merely a queue.** Work may begin as knowledge, remain knowledge, become a process, or become dispatchable implementation. Yesod preserves those transitions instead of forcing every observation into a ticket.

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

Part II.

From Intent to Merge

5 Planning and Dispatch

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

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

5.1. Capture Before Decomposition

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

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

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

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

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

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

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

3. add targeted tests and operator documentation.

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

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

5.2. Complexity Is Routing Data

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

Useful dimensions include:

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

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

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

5.3. Choose Merge Boundaries Before Task Order

The planner makes two different dependency decisions.

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

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

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

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

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

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

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

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

5.4. Pre-flight Questions

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

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

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

5.5. Arming

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

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

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

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

When the work must not run, the control plane can apply an explicit negative route:

```
yesod codefactory-dispatch hold NOTE_ID --reason "requires operator access"
```

The `hold` sentinel removes the note from ordinary claiming without pretending that implementation is complete or that code failure has blocked it.

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

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

6 The Runner

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

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

6.1. The Ten-Second Loop

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

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

Discovery then intersects two durable inputs:

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

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

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

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

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

6.2. The Claim Boundary

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

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

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

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

6.3. Worktree Isolation

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

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

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

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

6.4. Timers That Define Failure

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

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

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

6.5. The Run Ledger

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

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

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

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

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

7 Governed Agent Execution

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

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

7.1. The Worker Canon

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

The canon's main disciplines are:

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

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

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

7.2. Governance Around the Process

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

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

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

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

7.3. What Counts as Success

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

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

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

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

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

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

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

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

7.4. Failure Has Two Axes

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

Outcome verdicts describe the work product:

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

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

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

7.5. Bounded Retries

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

The rule behind these details is simple:

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

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

8 The Autonomous Merge Engine

The diagnostic change now exists as a verified candidate branch. That proves the worker produced retrievable code; it does not prove that the code composes with the current shared branch or deserves to join it. That decision belongs to the **Autonomous Merge Engine**, or **AME**.

In Yesod's vocabulary, each tool enters the integration plane through a **refinery profile**: the repository-specific contract that tells the AME where the code lives, how to gate it, and what to do after a merge. The runner proves a branch exists; the AME decides whether that branch may join shared history.

8.1. Supervisor and Ephemeral Worker

The AME separates continuous coordination from one-shot integration.

The **supervisor** is a long-running, deliberately small poller. It loads database-backed refinery profiles, reconciles the tools it owns, performs queue maintenance, chooses one tool in round-robin order, respects a global merge lock, and spawns a worker for one integration attempt.

The **ephemeral merge worker** does the mutating work. It resets a dedicated checkout to current `origin/main`, selects and verifies a batch, merges the candidates locally, runs the configured gate, records a green or red result, writes the merge job, and exits.

This split limits two common daemon hazards. Starting a fresh worker reloads merge logic for each attempt rather than trusting old in-memory code, and using a dedicated checkout prevents the AME from resetting a developer's working tree. Durable tables, not a supervisor's memory, carry state across worker crashes and restarts.

8.2. Multi-Repository Profiles

Refinery configuration is no longer a hardcoded Yesod-only table. Database profiles are authoritative, with source defaults available as a fallback. A profile can declare:

- tool and candidate-branch prefix;
- repository checkout;
- gate type and command;

- owned or upstream-contribution kind;
- deploy command, trigger paths, and action;
- fork and upstream remotes;
- deploy key.

At the deployment snapshot, the live registry contained twelve profiles covering Python, Node, and Rust repositories, plus shell-exit, delta-based, and upstream-contribution workflows. That is evidence of a functioning multi-repository factory rather than a Yesod-only mechanism with future-looking names.

One edge remains incomplete. The ephemeral worker has an upstream-contribution finish path that can gate a change, push it to a fork, and record a pull request. The always-on supervisor, however, excludes upstream profiles from autonomous draining. The integration mechanism exists; its unattended control-loop connection does not yet.

8.3. Queue Reconciliation as a Control Loop

The merge queue is durable working state, not the final record of truth. Durability lets integration survive a crash, but it also preserves stale and partially written rows. Reconciliation is the control loop that repeatedly compares queue state with notes, branches, retries, and merge history.

On each pass, it can:

- enroll eligible `awaiting_merge` notes;
- refuse branchless non-data enrollment;
- drop rows whose notes have left the merge lifecycle;
- deduplicate independent enqueues;
- remove ghosts whose commits already landed;
- detect reverted merge records;
- return `gate_red` rows below the retry cap to `queued`;
- block rows that reach the cap;
- bypass blocked rows so healthy work can pass;
- evict stale blocked rows already present on `main`;
- reopen sufficiently old retry-capped notes.

The final behavior concerns note lifecycle rather than queue status: a `blocked` note whose execution retry count reached the cap can reopen after a 120-minute recovery window with that count reset. Without a cause fingerprint or recovery counter, a deterministic defect can return in two-hour waves. Self-healing needs a bound of its own.

8.4. Batch Selection

The worker walks queue order and assembles a same-tool batch. Two candidates are considered incompatible when they overlap files, both modify contested schema machinery, or touch configured hot paths.

The policy is deliberately conservative. A false incompatibility sacrifices throughput for one cycle. A false compatibility can create a merge interaction that neither branch tested.

File conflicts also reveal architecture. If most changes touch one monolithic module, even logically independent work must serialize at integration. Faster gates reduce the immediate cost, but decomposition—not scheduler cleverness—is the durable fix.

8.5. Branch Verification

Before a candidate enters the batch, the worker fetches its remote branch and asks three questions:

1. Does the branch exist?
2. Is it ahead of current `origin/main`?
3. Is its diff empty because the work has already landed?

A missing branch is skipped at this late stage. Earlier enrollment guards and reconciliation make persistent missing branches less likely, but an already-enrolled row without a branch remains an important diagnostic condition: execution state claimed a candidate that Git cannot supply.

For valid candidates, the AME merges into its local checkout first. Only the resulting candidate tree—not each branch in isolation—is presented to the gate.

8.6. Green

A green gate is necessary, but the AME must still verify integration. It pushes the merged tree to `origin/main`, confirms the remote SHA, and checks that each note's commit is an ancestor of the new shared history. Only then does it mark notes `done` and write the durable merge summary. Here, `done` records source integration; it is not yet proof of runtime delivery.

The current source does not automatically delete the remote candidate branch. Older documentation that promises branch deletion overstates the implementation.

8.7. Red

A red batch never reaches `origin/main`. When a batch contains several notes, isolation tries to identify the offender while preserving innocent work that can still pass. The red queue row accumulates a durable attempt; the default queue retry cap is two.

Declared queue statuses include `gate_green`, `verifying`, and `merging`, but the current automated transitions do not use all of them as a detailed phase machine. For fine-grained progress, `merge_jobs`, `refinery_workers.phase`, and `gate_runs` are more reliable than interpreting the queue row alone.

8.8. Janitorial Layers

The phrase "the janitor" currently refers to three distinct layers:

1. **AME per-tick queue maintenance** actively repairs several deterministic queue conditions.
2. **The refinery janitor**, integrated in shadow mode by default, detects and records problems without mutating them.
3. **The factory retention library** can sweep worktrees, mail, and runner logs, but at the audit snapshot it was not wired into a live service or CLI.

The refinery janitor's vocabulary is specific. A **ghost note** still awaits merge even though its branch is already on `main`. A **phantom queue row** has lost its branch and no longer belongs to an `awaiting_merge` note. A **rogue supervisor** is an extra live supervisor beyond the one expected.

Broader cross-schema cleanup, threshold alerting, and verification aging existed in separate development work but were not all present as an operational mainline service. Keeping these layers distinct prevents "the janitor exists" from being mistaken for proof that every cleanup policy is deployed and active.

> **Invariant — Durable scratch must be reconciled.** A queue that survives crashes also survives stale rows. Persistence without reconciliation converts transient failure into permanent obstruction.

The AME supplies integration discipline, but its verdict is only as trustworthy as the environment that runs the gate. For Yesod's most demanding path, that environment is a disposable remote MicroVM.

9 The Remote Merge Gate

The merge gate evaluates the candidate tree that the AME assembled. It is the last pre-merge authority: workers may propose code and the AME may compose it, but only a green profile gate permits the automated push to `main`.

Yesod supports both local and remote gate backends. For Yesod itself, the fast pre-merge tier is:

```
cd yesod && uv run pytest -q -p no:cacheprovider -n auto \
  -m 'not bd_integration'
```

The slower `bd_integration` tier runs after merge through CI and is recorded by `yesod main-ci`. It remains observable and can produce warnings, but it is not a pre-merge blocker. Older descriptions that call the fast merge command the complete suite are stale.

9.1. Why Move the Gate

A local gate consumes the same CPU, memory, and filesystem environment used by the operator and merge process. Red-batch isolation compounds the problem because each candidate may need another serial run. A slow or contaminated local environment therefore reduces throughput and weakens confidence at the same time.

The remote backend moves the same gate into a disposable MicroVM with its own repository checkout and PostgreSQL instance:

The ephemeral merge worker sends a batch specification naming the base and candidate branches to a **loopback pool proxy**—a host-local service that owns the cloud credentials and MicroVM lifecycle. The proxy injects authentication, lazily starts or resumes a VM, forwards the gate request, and later suspends or reaps the instance. Inside the VM, `gate_server` fetches the branches, constructs the candidate tree, runs the gate, and returns a structured verdict.

Because the executor itself is disposable, gate telemetry is mirrored into shared PostgreSQL. Operators can observe the run without relying on the continued existence of the VM that produced it.

9.2. Verdict Versus Infrastructure

The most important remote-gate distinction is whether the candidate failed or the gate failed to run.

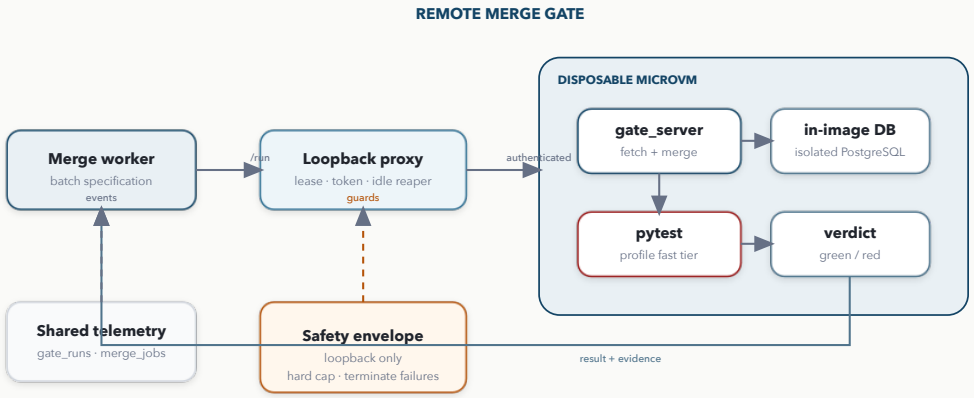

Figure 9.1.: The MicroVM merge-gate path and its safety envelope.

A red gate result—whether from failing tests or an error constructing the candidate tree—is a **verdict**. It says the candidate was evaluated and rejected. The result consumes a gate attempt and enters the AME's red-handling path.

A transport error, timeout, unavailable proxy, or failed remote-gate hook is **infrastructure failure**. No trustworthy judgment about the candidate exists. The remote client tries the backend at most twice, invalidating its warm-pool state after the first failure. If the second call also fails, it raises `GateInfraError`. Because no verdict exists, the failure neither consumes the AME's durable gate attempt nor triggers a local fallback; the work remains queued.

> **Design Trade-off — No fallback.** Local fallback improves availability but silently changes isolation, performance, and resource assumptions during an outage. Yesod chooses visible stalled integration over an implicit backend change.

The July 11 loopback guard makes that boundary fail closed. A configured remote-gate URL must point to the local proxy, and a direct cloud runtime URL is refused. Rejecting the URL does not quietly disable remote mode; without a valid pool, the failure surfaces as gate infrastructure trouble rather than rerouting tests to the local machine.

9.3. The Launch-Storm Incident

Disposable infrastructure creates a dangerous asymmetry: cleanup can fail silently while retry creates another resource.

On July 10 and 11, false image-identity changes made the pool believe it needed fresh capacity, while a proxy lock-ownership bug allowed launches to escape the intended single-instance lifecycle. Retries multiplied the leak. Dozens of orphaned MicroVMs accumulated before the pattern was contained.

The response became a layered safety design:

- normalize `None`, empty, and whitespace image identities;

- track pool warmth independently of the image string;
- invalidate state explicitly before a fresh retry;
- serialize launches;
- make `_launch()` own its lock correctly;
- terminate an instance that fails before adoption;
- expose explicit `/shutdown` for owned isolation instances;
- reap shared instances after five idle minutes;
- refuse non-loopback remote URLs;
- disable parallel red isolation by default;
- configure a hard fleet launch cap of three;
- retain a separate high-water tripwire for alerts.

No single item is the whole fix. Normalization prevents false change detection; locking prevents concurrent creation; ownership cleanup closes the failed-adoption leak; the hard cap is intended to limit damage if earlier assumptions fail; and the tripwire is intended to make an abnormal fleet visible. Both fleet-wide controls still depend on the counting contract examined below.

> **Factory Floor — Bound resource creation before retrying.** A retry loop around a leaky create operation is a cost amplifier. The hard cap must sit before creation, and failed adoption must terminate the resource it just created.

9.4. A Critical Integration Finding

A safety mechanism is only as strong as the live API contract it actually reads. The source audit found that the pool proxy's `_count_running_microvms()` expects the response key `microvms`, while sibling fleet-poller code expects `items`. Unit tests mock `microvms`.

If the production API returns `items`, the proxy can count zero even while instances are running. Both the hard cap and the high-water tripwire would then be present in source yet ineffective in operation. The count also fails open when the fleet-list request raises an API error, effectively treating an unknown fleet size as safe.

This is why the existence of a constant or passing mock test is not enough. The fleet guard must be verified against a real service response, including the failure behavior.

There is a separate policy question about scope. The current count is fleet-wide rather than filtered to the gate image. Once the response shape is verified, unrelated MicroVM workloads could exhaust the limit and deny gate launches. Safety and availability therefore require an explicit decision about which fleet the cap protects.

9.5. Parallel Isolation Is Off

Earlier documentation advertised ten-way remote red isolation. The pinned source keeps it disabled unless `YESOD_GATE_PARALLEL_ISOLATE=1` is set.

That default matches the current single-instance proxy. Per-note isolation pools would share one proxy, and each owned pool would call `/shutdown`. One note could therefore terminate

the MicroVM while another still depended on it. Safe parallel isolation requires distinct per-shard proxy URLs or another lease model.

The code path's existence is not evidence that its resource-ownership model is safe. Do not enable the flag on that basis alone.

9.6. Frozen Images

The VM image is deployed code. Merging `gate_server.py` to `main` does not update an image that has already been built.

The current rebuild check compares dependency-lock and schema information but does not comprehensively hash the gate server and Dockerfile recipe. Repository truth can therefore advance while the executable gate environment remains frozen on older code.

Every image should carry a source SHA and recipe digest, and every gate result should record both. Without that chain, a remote gate may be isolated and repeatable yet still leave operators unable to answer the simplest provenance question: **which gate code produced this verdict?**

A trustworthy green verdict permits the AME to advance Git. It still does not prove that anyone is running the merged code.

10 Merge Is Not Deploy

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

For the diagnostic-command example, a green merge means the command's source is present on `origin/main`. The installed `yesod` CLI—or a long-lived service that exposes related behavior—may still be using an older checkout. A deployment hook may also have skipped the changed path or failed to restart a service. The note can already be `done` in the source lifecycle while the feature remains unavailable to users.

10.1. Profile-Driven Hooks

After a successful push, the AME derives a deploy action from the tool's profile and the paths changed by the batch. A profile can choose `none`, `static`, `restart`, or `custom`, with optional path prefixes that trigger the action.

The post-merge hook executes from the repository at the merged SHA. It writes standard output and error to a per-run log and returns one structured outcome:

- `deployed` — the configured command exited successfully, though this is not independent runtime proof;
- `skipped` — no action or command applied;
- `failed` — the command could not start or returned nonzero.

The merge job retains the outcome and any hook log path. A failed hook does not roll back the merge: shared Git history has already advanced, and rewriting it would create a second incident. Instead, the factory records and escalates the deployment failure.

This corrects two tempting simplifications. A merge is not guaranteed to deploy because hooks can skip or fail, and deployment is not a Yesod-specific hardcoded action. It is part of each tool's refinery contract.

10.2. The Yesod Runtime Path

For Yesod, source changes can require a service restart, while static visualization changes may need only asset delivery. The current deployment script targets the runtime host and reports service health.

One weakness remains: the health script in the inspected deployment path echoes expected HTTP codes rather than asserting every code as a blocking condition. A hook can therefore

look successful even when its service-level check is weaker than its name suggests. Post-merge verification should prove the running version and behavior, not merely prove that a reassuringly named script exited zero.

10.3. Checkout Roles

The deployment snapshot contained three local checkouts with different owners and purposes:

- the **development checkout**, where new source was inspected;
- the **services clone**, which supplied the installed runtime used by long-lived local processes;
- the **AME workdir**, which the merge worker resets and mutates.

This separation is healthy when it is explicit. It keeps destructive merge operations away from a human's working tree and gives long-lived services a stable installation source. It becomes dangerous when an operator assumes the development checkout controls a daemon that imports from the services clone, or when a merge worker is accidentally pointed at a human checkout.

The authority rule is straightforward: the AME must have scratch authority over its workdir; it must never treat the operator's checkout as disposable.

10.4. Observed Fleet Topology

The following diagram records a dated deployment observation, not a guarantee made by the source repository.

At that snapshot:

- `pompom` hosted a local runner, the AME supervisor, loopback gate proxy, gate-fleet poller, mail and status bridges, and `opencode serve`;
- three dedicated Linux runner VMs and a dispatch host ran system services;
- PostgreSQL 17 and Dolt SQL ran on dedicated VMs under the `seykhl` hypervisor;
- `dertog` hosted `yesod serve`, the live-viz watcher and WebSocket service, and the propulsion service;
- the gate path reached disposable cloud MicroVMs through the loopback proxy.

The inspection also exposed several seams:

- manual guard and shell processes, rather than fully declared launchd units, protected the AME and gate proxy;
- deployed guard behavior differed from the repository version after the launch storm;
- runner checkouts lagged current `main` even while their services appeared healthy;
- the software propulsion system (SPS) ran on `dertog` while the AME ran on `pompom`.

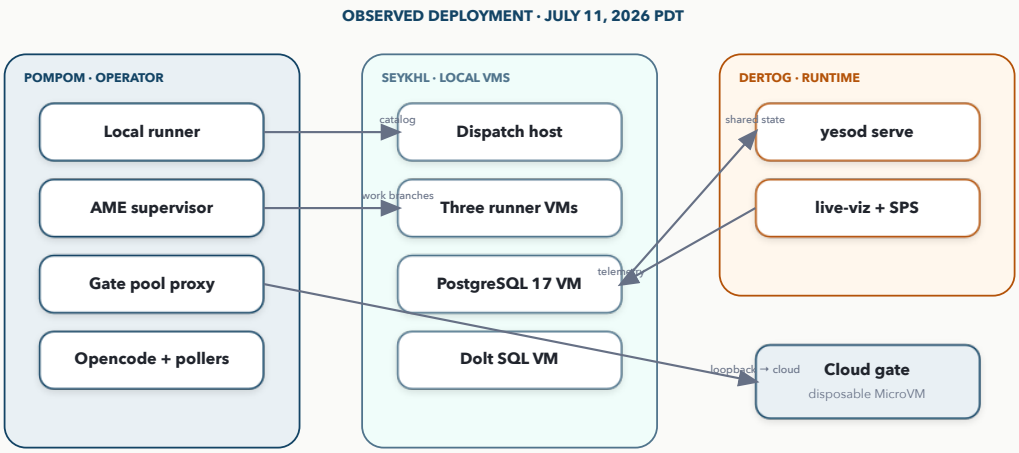

Figure 10.1.: Observed Yesod topology on July 11, 2026 PDT.

The final point changes the meaning of a PID. Parts of the SPS watchdog use local PID signals and local child or file liveness checks against the AME's recorded PID. Across hosts, that PID may identify no process—or, worse, an unrelated one. The 60-minute emergency-stop path carries the same local-signal assumption.

> **Factory Floor — Host placement is part of correctness.** A watchdog that calls `os.kill()` is not host-agnostic merely because it reads the target PID from a shared database. Either co-locate it, route the signal explicitly, or make the host mismatch a hard refusal.

10.5. Proving Delivery

A robust delivery proof connects four points of evidence:

1. the candidate note and branch;
2. the merged `origin/main` SHA;
3. the post-merge action and outcome recorded for that SHA;
4. the version or artifact digest and health evidence reported by the runtime.

The first two establish source integration. When the hook actually deploys, linking the third and fourth shows that deployment acted on the same history and that the matching runtime is healthy. A skipped or failed hook instead records why delivery has not been demonstrated. Until the chain is complete, `done` may mean source-complete rather than user-visible.

This closes the journey from intent to merge—and reveals why merge cannot be the end of the operational story. The next part turns from moving one change through the factory to observing and operating the factory as a whole.

Part III.

Operating the Factory

11 Seeing the Whole System

Yesod is observable through several surfaces because no single table contains the whole truth.

The operator's job is not to find one perfect dashboard. It is to select the evidence surface that owns the question.

11.1. The Evidence Map

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_runs`, gate log, verdict
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

The table encodes a discipline: prefer the narrowest authoritative source. A fleet page may summarize queue depth, but the queue row owns the integration status. A note page may show a result commit, but Git owns ancestry.

11.2. Command-Line Windows

The current CLI provides read-only summaries for each plane:

```
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 refinery profile list
yesod main-ci status
yesod agent roster
```

The commands are intentionally composable. `factory status` is a high-level pulse; `runs show` is an attempt autopsy; `refinery show` is an integration autopsy.

`yesod daemon-sweep` inventories local daemon-like processes against expected units. It is read-only, which makes it safe to use during diagnosis. Its output still requires interpretation: a new legitimate daemon can look rogue until the expected-process registry learns it.

11.3. The Runs Surface

The run ledger opens at claim and finalizes once. The web and CLI surfaces expose:

- note and bead identity;
- lane, model, host, and runner;
- start and finish times;
- status and exit code;
- tokens and cost;
- result commit;
- acceptance result;
- verdict and failure classification;
- routing rationale;
- archived log and progress events.

Progress heartbeats are stored separately in `dispatch_log` so an operator can see incremental narration while the main process buffers output. The event feed is useful for liveness, but the finalized run row remains the attempt ledger.

11.4. The Refinery Surface

Integration uses several records with different retention:

- `merge_queue` — mutable working state;
- `refinery_workers` — supervisor and ephemeral-worker heartbeats/phases;
- `merge_jobs` — one durable record per integration attempt;

- `gate_runs` — gate progress and verdict history;
- `merge_summaries` — durable record of landed work;
- `refinery_stalls` — janitor and stall observations;
- `refinery_gate_fleet` — current MicroVM fleet snapshot.

The live API exposes current and recent gates, queue rows, jobs, fleet state, throughput, and recent merges. The visualization layer can reconstruct a useful picture after reload because progress is persisted rather than held only in a browser stream.

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

11.5. Passive Observation Must Stay Passive

The gate proxy distinguishes activity endpoints from passive health and status endpoints. A fleet poller may ask for status every few seconds without launching a MicroVM or resetting its idle timer.

This is a subtle but essential observability property. A monitoring system that changes the resource it observes can keep idle infrastructure alive forever, mask demand behavior, or trigger the very launch storm it is meant to detect.

11.6. Dated Deployment Facts

The live fleet is part of the evidence map, but it is not versioned by the source repository. The audit therefore records deployment facts with a date.

On July 11, the operator plane, runner VMs, dedicated data VMs, runtime host, and cloud gate were reachable and their primary services were running. The same inspection also found topology drift and stale checkouts. Both facts can be true: a service may be healthy while its deployment discipline is weak.

This is why the authoritative documentation project labels deployment claims separately. Source paths can drift by line number; running processes can drift without any commit at all.

11.7. A Read-Only Diagnostic Rhythm

When something appears stuck, use a fixed rhythm:

1. identify the note and linked bead;
2. inspect note status, lane, hold, retry time, and dependencies;
3. inspect Beads readiness and blocker integration;
4. inspect runner eligibility and live attempts;
5. inspect the remote branch;

6. inspect queue row and merge job;
7. inspect gate progress or verdict;
8. inspect `origin/main` ancestry;
9. inspect deploy outcome and runtime version.

The order follows the pipeline. Jumping directly to restarting a daemon skips the evidence that says whether the daemon is the problem.

> **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

Yesod's reliability is not the absence of failure. It is the conversion of common failure into bounded, diagnosable transitions.

12.1. Execution Failures

Dead runner. A stale runner heartbeat can release a claim, but the recovery path checks for a live run first. This avoids reclaiming work whose daemon heartbeat is late while the worker is still progressing.

No progress. Two consecutive no-progress verdicts halt ordinary retries and block the note. The control prevents an evidence-free hope loop.

Partial or uncommitted work. The worktree is retained and the verdict records that something useful may exist. Triage can recover the edits or write a better follow-up task.

Verification failure. A branch with commits but an explicit failing acceptance or TUI check is parked honestly. It is not manufactured into `awaiting_merge`.

Push failure. Remote branch proof happens before integration. If the branch cannot be confirmed, the note remains outside the merge path.

Rate limit or tool outage. Infrastructure classification can release the note without consuming an ordinary model retry, separating "the model failed" from "the lane could not run."

12.2. Integration Failures

Queue enrollment gap. The note reaches `awaiting_merge` but no row is written. Reconciliation can enroll it later if the branch is valid.

Missing branch. New branchless non-data enrollment is refused. Existing rows are still diagnosed through branch verification and janitor logic.

Ghost branch. An empty diff or commits already on `main` can be removed from the queue and reconciled to the landed state.

Red gate. A real verdict increments the durable attempt, isolates multi-note batches, and blocks at the cap.

Gate infrastructure failure. No verdict means no gate attempt. The worker exits with infrastructure evidence and leaves the work queued.

Blocked head item. Active items can be reordered ahead of blocked rows. A single broken branch should not stop the entire factory.

Stale blocked item. If its patches are already on `main`, it can be evicted and completed. If it remains blocked past the recovery window, current code can reopen it—useful for transient failure, risky for deterministic defects.

12.3. Service Failures

The SPS has two recovery layers in current source.

The first is a roughly 16-minute AME watchdog. It distinguishes flowing work, all-blocked queues, and stalled queues. It suppresses unsafe kills when blocked rows exist or liveness signals show a gate is active.

The second is a 60-minute emergency mode based on an old awaiting-merge item plus lack of recent merges. It can stop AMEs and hand work to the mayor while telling ERS to stand down.

Those layers are more sophisticated than the old "kill and restart" description, but deployment placement creates a serious caveat. At the audit snapshot, SPS ran on `dertog` and the AME on `pompom`. Local `os.kill()` and local process/file checks cannot safely control a remote PID. The same numeric PID can refer to no process or the wrong process on another host.

The correct fix is architectural: make the control operation host-addressed, enforce co-location, or refuse to signal when the recorded host does not match the local host.

12.4. Chronic Degradation

Acute failures are usually visible. Chronic degradation is harder:

- old worktrees consume disk;
- local-backend tests can leak schemas;
- logs grow;
- stale images drift from source;
- runner checkouts fall behind;
- manual guards diverge from declared service units;
- canceled or red post-merge CI has no automatic remediation actor;
- deploy health can be reported more strongly than it is asserted.

Some cleanup functions exist as library code; some queue maintenance is live; some alert work exists outside mainline production. The operational question is always: **what process is scheduled, on which host, from which source checkout?**

12.5. Incident Decision Matrix

Symptom	First evidence	Automatic response	Human next step
Armed note never claims	note + bead + runner eligibility	none beyond polling	correct link, staging, pin, lane, or dependency
Run stops narrating	run heartbeat/log + process	runtime/kill governance	preserve worktree, classify cause
Repeated no-progress	run verdict history	block after two	rewrite scope or change lane after triage
Awaiting merge, no row	branch + reconcile report	reconcile enrollment	repair branch/status if invalid
Queue row, missing branch	Git remote + queue	guard/janitor detection	recover or reset with evidence
Gate red	gate verdict/log	isolation + capped retry	fix code/test, then re-enter
Gate unavailable	proxy/worker events	two infra retries, leave queued	restore backend; do not burn gate attempts
Blocked item stalls queue	queue positions	bypass blocked	triage offender
Merge landed, deploy failed	merge job + hook log	notification/flag	repair runtime without rewriting merge
Pipeline stall alarm	host-aware service evidence	guarded watchdog	verify placement before signaling

12.6. Recovery Rules

Three rules prevent a recovery from becoming a second incident.

Change something before retrying. Fix the defect, repair infrastructure, strengthen the lane, or narrow the task.

Do not erase the only evidence. A retained worktree, gate tail, branch, or queue scratch field may be the only trace of the failure.

Respect authority and ownership. Workers do not merge. Triage prepares destructive commands rather than improvising them. Root-bead closure belongs to terminal synchronization, not an eager child worker.

> **Factory Floor — Maturity is scar tissue.** The valuable hardenings are recognizable because each encodes a failure the system refuses to repeat: false green, runner masquerade, branchless merge enrollment, gate restart loops, orphaned MicroVMs, and evidence-free retries.

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.

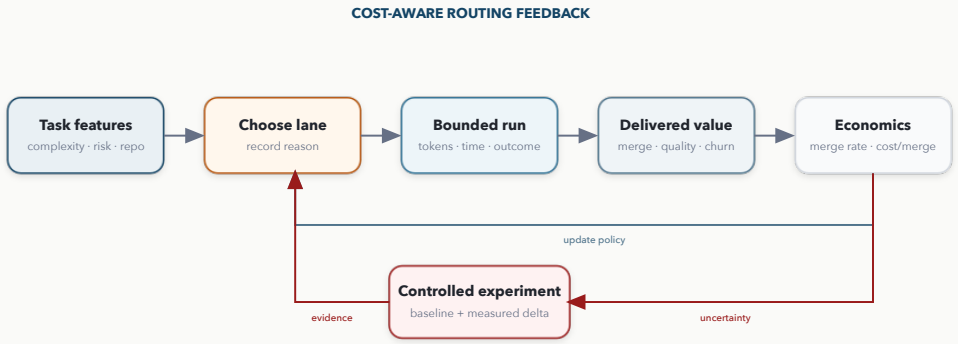

Figure 13.1.: The cost-aware routing feedback loop.

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 IV.

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 supervisor. 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 Future of Yesod

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

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

15.1. 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?"

15.2. 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.

15.3. 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.

15.4. An Adaptive Dispatch Economist

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

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

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

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

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

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

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

15.5. 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.

15.6. 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.

15.7. 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.

15.8. 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.

15.9. 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.

15.10. 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.

15.11. 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.

15.12. A Plausible Sequence

The future can be staged.

Near term: verify the MicroVM fleet-count integration, make watchdog actions host-addressed, declare the AME and proxy as managed services, wire retention cleanup, strengthen deploy/runtime proof, and close the image-provenance gap.

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

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

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

Epilogue: Scar Tissue

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

The impressive part is the accumulation of refusals.

Do not call an exit code a commit. Do not call a local commit a remote branch. Do not call a branch a merge. Do not call a merge a deployment. Do not call a transport failure a red test. Do not call an unknown price zero. Do not call an agent's claimed identity authority. Do not retry forever. Do not launch forever. Do not let one blocked row stop unrelated work. Do not let a watchdog kill a healthy gate. Do not let a process touch a checkout it does not own.

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

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

A Command Field Guide

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

A.1. Catalog and Knowledge

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

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

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

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

A.2. Processes and Molecules

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

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

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

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

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

A.3. Work Graphs

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

```
yesod bd TOOL -- show BEAD_ID
yesod bd TOOL -- children BEAD_ID
yesod bd TOOL -- dep list BEAD_ID
yesod bd TOOL -- graph BEAD_ID
yesod bd TOOL -- ready --json
```

A.4. Factory Status

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

A.5. Dispatch and Runs

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

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

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

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

A.6. Refinery

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

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

A.7. Propulsion

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

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

B States and Timers

B.1. Note State

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

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

B.2. Dispatch State

`agent_dispatch` is orthogonal to lifecycle:

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

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

B.3. Run State

The run ledger admits:

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

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

B.4. Merge Queue State

Declared values are:

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

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

B.5. Timing Reference

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

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

C Roles and Authority

C.1. Agent Roles

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

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

C.2. Deterministic Services

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

C.3. Authority Rules

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

D Evidence and Source Map

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

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

Claim	Source anchor
Note lifecycle declaration	`yesod/src/yesod/note_status.py`
Catalog schema and audit tables	`yesod/src/yesod/db.py`
Audited status/dispatch writes	`yesod/src/yesod/note_status_writer.py`
Agent role primers and authority	`yesod/src/yesod/prime.py`
Shared worker instructions	`yesod/src/yesod/worker_canon.py`
Runner claim and governance	`yesod/src/yesod/runner.py:Runner`
Outcome reconciliation	`yesod/src/yesod/reconciliation.py`
Failure classification	`yesod/src/yesod/failure_analysis.py`
Run ledger and rollups	`yesod/src/yesod/run_ledger.py`
Lane registry	`yesod/src/yesod/dispatcher/runners.py`
Usage and pricing	`yesod/src/yesod/dispatcher/usage.py`
Processes and molecule projection	`yesod/src/yesod/processes.py`
Cross-catalog question answering	`yesod/src/yesod/ask.py`, `query.py`
Queue reconciliation	`yesod/src/yesod/refinery_queue.py`
AME supervisor	`yesod/src/yesod/refinery_supervisor.py`
Ephemeral merge worker	`yesod/src/yesod/ephemeral_worker.py`
Batch/deploy decisions	`yesod/src/yesod/merge_core.py`
DB-backed profiles	`yesod/src/yesod/refinery_profiles.py`
Post-merge hook	`yesod/src/yesod/post_merge_hook.py`
Gate telemetry	`yesod/src/yesod/gate_telemetry.py`
MicroVM proxy	`yesod/docker/microvm/gate-pool-proxy.py`
Cost/quality statistics	`yesod/src/yesod/stats_data.py`
Propulsion and watchdogs	`yesod/src/yesod/propulsion_service.py`
Queue janitor	`yesod/src/yesod/refinery_janitor.py`
Retention library	`yesod/src/yesod/factory_janitor.py`

D.1. Current Deployment Snapshot

Observed July 11, 2026 PDT:

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

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

D.2. Documentation Corrections Incorporated Here

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

- multi-repository refinery profiles are live, not merely planned;
- runtime compatibility for exact child notes still exists, but current planner guidance uses one feature note and root Bead as the merge unit;
- planners leave completed feature notes unarmed; dispatchers own lane choice and arming;
- Bead dependencies order work inside one feature, while note dependencies order features across merge boundaries;
- arming has several transports, not only one HTTP endpoint;
- the automated note path does not normally write `merging`;
- status advance and queue enrollment are separate durable operations;
- optional skipped checks are not explicit failures for ordinary code tasks;
- successful worktrees are cleaned; failed/unverified worktrees remain;
- merged remote branches are not automatically deleted by the current path;
- deploy is profile-driven and may skip or fail after merge;
- remote red isolation is sequential by default;
- July 11 storm controls include loopback refusal, failed-launch termination, explicit shutdown, corrected lock ownership, and a hard cap;
- SPS now has guarded and emergency recovery layers, though live cross-host placement undermines local PID signaling;
- janitorial capability is layered: active queue maintenance, shadow refinery janitor, and an unwired retention library;
- threshold coverage remains incomplete despite richer SPS findings;
- the MicroVM hard-cap response-key integration was subsequently verified in live service during the self-hosting exit wave.

E Yesod Development Waves

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

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

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

E.1.1. Wave Summary

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

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

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

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

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

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

E.1.2. The Cutoff Was a Feature

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

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

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

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

E.1.3. What Shipped

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

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

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

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

E.1.4. A MicroVM Became a Managed Gate

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

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

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

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

E.1.5. Evidence at the Exit

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

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

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

E.1.6. Ready Does Not Mean Configuration-Free

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

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

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

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

E.1.7. The New Metric

The exit criterion changes what Yesod should optimize.

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

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

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

E.2.1. Wave Summary

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

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

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

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

E.2.2. The Gate Became a Job Protocol

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

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

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

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

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

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

E.2.3. The Cutover Failed at the Image Boundary

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

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

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

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

E.2.4. The Operator Stopped Needing `psql`

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

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

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

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

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

E.2.5. Observability Learned to Name States

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

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

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

E.2.6. What Shipped, and What Stayed Out

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

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

E.2.7. The New Exit Criterion

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

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

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

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

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

E.3.1. Wave Summary

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

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

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

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

E.3.2. Readiness Became a Control-Plane Fact

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

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

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

E.3.3. The Refinery Became Crash-Only

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

The new lifecycle is built from several cooperating mechanisms:

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

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

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

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

E.3.4. The Fleet Survived Its First Real Storm

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

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

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

E.3.5. A Generic Gate Acquired a Repository Contract

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

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

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

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

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

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

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

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

E.3.7. The Dashboard Became a Deployable Surface

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

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

E.3.8. What Shipped, and What Stayed Out

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

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

E.3.9. The New Exit Criterion

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

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

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

E.4.1. Wave Summary

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

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

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

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

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

E.4.2. A Note Can Choose Its Integration Branch

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

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

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

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

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

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

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

E.4.3. `main` Stayed Releasable

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

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

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

E.4.4. Distillation Became a Bounded Maintenance Job

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

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

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

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

E.4.5. A Stall Now Pages a Human

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

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

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

E.4.6. The Proxied Dashboard Became Complete

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

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

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

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

E.4.7. What Shipped, and What Stayed Out

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

E.4.8. The New Exit Criterion

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

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

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

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

E.5.1. Wave Summary

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

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

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

E.5.2. The Demo Had a Real Dependency Shape

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

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

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

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

E.5.3. Dispatch Became Observable Product Behavior

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

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

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

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

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

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

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

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

E.5.5. Operations Were Part of the Proof

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

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

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

E.5.6. What Shipped, and What Stayed Out

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

E.5.7. The New Exit Criterion

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

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

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

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

E.6.1. Wave Summary

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

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

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

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

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

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

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

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

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

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

E.6.3. Coordination Has a Lifetime

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

Wave 6 therefore assigns each information surface a lifetime:

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

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

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

E.6.4. The ALS Incident Exposed an Identity Boundary

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

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

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

Recovery followed the authority boundary in the safe direction:

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

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

E.6.5. The Server Became Canonical

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

The branch carries the rule through the whole lifecycle:

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

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

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

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

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

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

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

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

E.6.7. What Has Landed, and What Remains

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

E.6.8. The New Exit Criterion

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

The wave is complete only when two proofs exist. First, every server-backed workspace must derive identity from a known database of record, while every clone-local guard stays outside Git and every failed note linkage remains recoverable. Second, an idle factory must be able to explain that it is idle, wait through a bounded grace period, stop every autonomous component exactly once, remove transient coordination messages, and remain quiet until a human explicitly starts a clean session.

The durable lesson is that autonomy needs an off-state as carefully designed as its run-state. A factory that cannot distinguish durable truth from transient coordination will either forget important work or preserve activity forever. The quiet factory does neither.

E.7. Wave 7: The Legible Factory

E.7.1. Wave Summary

Field	Value
purpose	make the factory understandable as a chronological, filterable story; distinguish work that is rejected from work that is accepted but intentionally parked; preserve grounded plans without accidentally scheduling them
development window	opened August 7, 2026; this entry records the opening checkpoint rather than a completed exit
baseline	`162de6dd` on `main`
triggering friction	an AWS-hosted roadmap produced executable plans that were not ready to run; Yesod had no `deferred` lifecycle state; the existing `/viz/live-events.html` page exposed a narrow event rail rather than a human-scale activity stream
shipped foundation	catalog activity history, note-status history, a standalone live-events page, cursor-based event backfill, and the `yesod factory status` rollup
open work	`ys-yes-b3iy` adds the `deferred` lifecycle state; `ys-yes-xqhi` turns the live-events prototype into a leveled, multi-tool operator stream
exit condition	from one page, an operator can tell what happened, at the desired level of detail, for selected tools; accepted parked work is represented as `deferred`, remains non-dispatchable, and can return to `open` with its plan and history intact

Wave 6 asked whether the factory could become quiet without losing durable truth. Wave 7 asks the complementary question: **when the factory is active, can a human under-**

stand what it is doing without reconstructing the answer from its databases and dashboards?

Yesod already records more state than most software factories. Notes have lifecycle histories. Beads record executable structure and dependencies. Dispatch runs, leases, gates, merge jobs, agents, and mail each expose their own transitions. The problem is not missing state in the abstract. The problem is that state is distributed across surfaces with different vocabularies, retention rules, and levels of detail.

A human does not experience those surfaces as separate subsystems. They ask what just happened, what is running, what is waiting, what was deliberately put aside, and whether a transition matters. Wave 7 treats those questions as a product contract. The factory becomes legible when its durable records form a coherent story at the level of human attention.

This opening checkpoint does not declare Wave 6 complete. Identity hardening, idle shutdown, and transient-message cleanup retain their own exit obligations. Wave 7 begins from the state they helped clarify: durable truth and disposable execution are different things, and an operator surface must say which kind of evidence it is showing.

E.7.2. The Factory Had State but No Story

The command `yesod factory status`, tracked by `ys-yes-q0e7`, consolidated an important set of operational facts. It made queue, supervisor, runner, gate, and fleet state available through one rollup rather than a sequence of private queries. That is a major improvement in observability, but a status snapshot answers only one class of question: what appears to be true now?

An operator also needs causality. Why did a note leave `open`? Which Beads were created when it was grounded? Did a worker claim the task, or did a dependency change make it eligible? Is a gate merely slow, actively running, retrying an infrastructure failure, or finished red? A snapshot can show the latest state while concealing the transitions that explain it.

Yesod's current interfaces make that explanation possible but expensive. The catalog landing page has an activity stream for catalog mutations. Note detail can show lifecycle history. The live-viz event log carries Bead and dependency events. Refineries and run pages expose gate and execution timelines. An operator can assemble the story, but only by knowing which surface owns each piece and mentally joining their identifiers.

Legibility is the removal of that mental join. It does not mean flattening all events into an undifferentiated log. It means giving each event a stable identity, time, subject, actor, tool, kind, and human-readable meaning, then allowing the operator to choose how much machinery to see.

E.7.3. Parked Work Is Not Rejected Work

The AWS-hosted Yesod roadmap exposed a second vocabulary gap. Phase 1 was decomposed into four fresh feature records:

- external PostgreSQL portability (`ys-yes-9xio`, root Bead `yes-fq93`);

- authenticated HTTP mutation boundaries (`ys-yes-v4cl`, `yes-ozj5`);
- provider-isolated MicroVM runtime flavors (`ys-yes-zkae`, `yes-0isq`); and
- a clean-install acceptance path (`ys-yes-81wu`, `yes-pip6`).

Each root has three file-scoped child Beads, an internal dependency chain, acceptance criteria, and a targeted gate. The first three features can proceed independently; the clean-install proof depends on all three. This is executable planning, not a list of aspirations.

It is also work that should not run yet. The roadmap is exploratory, the AWS delivery decision has not been made, and placing twelve well-specified child tasks in front of an autonomous dispatcher would confuse preparedness with authorization.

Yesod's lifecycle vocabulary had no honest state for that combination. `open` means actionable work available for planning or dispatch. `wontfix` means the work was rejected or abandoned. A dependency-derived block means the work would proceed if its prerequisites completed. A hold or missing arm is an execution-control fact. None means "we accept this direction, have preserved a good plan, and intentionally do not want it scheduled."

Until a better state exists, the four AWS records are temporarily `wontfix`, with comments saying they should become `deferred` once `ys-yes-b3iy` ships. Their root Beads are closed, while the child trees, tags, dependencies, and plans remain preserved. The workaround is safe for dispatch, but semantically wrong. A future reader counting rejected work cannot distinguish an abandoned idea from a deliberately parked roadmap phase without reading the comments.

The proposed `deferred` state repairs that meaning. It represents accepted work that is intentionally outside the active backlog. It must never be claimable, dispatchable, or automatically armed. Returning it to `open` must preserve its tags, Bead linkage, dependency graph, metadata, plan, and history. `wontfix` then regains a precise meaning: no current intention to implement.

E.7.4. Grounding Became a Durable Asset

The temporary AWS state reveals a broader design principle: planning and scheduling are separate authorities.

A grounded feature is valuable before a worker is allowed to touch it. The note records the product boundary and acceptance test. Its root Bead gives the feature one merge identity. Three to ten child Beads give an agent concrete, file-level steps. Bead dependencies order work inside that merge boundary; note dependencies order independently mergeable features. Together they make the plan inspectable, reusable, and cheap to revalidate later.

None of that implies permission to spend tokens, launch a MicroVM, write a branch, or alter AWS. Grounding answers "what would we build?" Dispatch answers "should the factory build it now?" Conflating them makes careful roadmap work dangerous: the better the plan, the more likely an autonomous system is to mistake it for an immediate command.

`deferred` turns a grounded plan into a durable portfolio asset. The plan can survive a change in priorities, a missing provider capability, or a budget decision without polluting the active queue or being mislabeled as rejected. When it is revived, a planner does not start over. The

planner revalidates the baseline, file paths, dependencies, and acceptance gate, then returns the note to `open`.

E.7.5. From Event Rail to Activity Stream

The existing page at `/viz/live-events.html` is the natural surface for the other half of this wave. Its source, `yesod/live-viz/live-events.html`, is a small React page backed by `All-BeadsAdapter`. It loads snapshot events, opens a live stream, and renders a newest-first rail. The page recognizes generic audit events, Bead events, and dependency events. It shows a connection indicator, relative and wall-clock time, actor, target, and a short message.

That is useful infrastructure, but it is still a prototype rather than a factory activity product. It has no multi-tool selector, no event-kind filter, no expandable structured details, and no distinction between a milestone and a heartbeat. It keeps a client-side window of 500 events. Its pause control stops the relative-time clock but not event ingestion. Its typed-event handler currently risks appending the same event twice. A demo-data toggle remains in the production page.

The feature request `ys-yes-xqhi` does not call for another page. It names this page as the implementation target and asks the existing catalog, status, live-viz, Bead, refinery, and gate sources to converge on a normalized event envelope. Initial history and live delivery must use the same identity and filter semantics. Cursor recovery must make retention gaps visible rather than silently presenting an incomplete story.

The product distinction is selectable detail:

Level	Operator meaning	Representative events
high	meaningful milestones	tool onboarded; note created; every note lifecycle change; dispatch, gate, merge, deployment, or major failure
medium	work structure and decisions	root or child Bead created, claimed, or closed; dependency changed; planning completed; runner assigned; gate batch formed
low	diagnostic mechanics	heartbeat, retry, lease, queue transition, reconciler pass, worker phase, or per-shard gate progress

The levels are cumulative. High shows only milestones. Medium adds the work structure that explains them. Low adds the machinery needed for diagnosis. This is not merely a severity filter: a successful heartbeat is low-detail but not low-importance when diagnosing liveness,

while a routine note transition is high-level because it changes the user's understanding of the work.

The second control is scope. An All Notes-style multi-select tool filter should let the operator watch one repository, a related group, or the whole factory. The same filter must constrain loaded history and new streamed events. A newest-first catch-up view should coexist with a live-follow mode that does not steal scroll position while the operator is inspecting older evidence.

E.7.6. Attention Became a Control Plane

Raw telemetry optimizes for completeness. Human attention does not. It is bounded, changes with the task, and is easily defeated by an event source that treats every database write as equally prominent.

The high, medium, and low contract is therefore a control-plane decision. It defines which transitions deserve immediate human awareness and which remain available when the operator asks for more resolution. The classification must be canonical per event kind rather than guessed independently by each page. Otherwise a gate retry will be a milestone in one view, a diagnostic in another, and absent from a third.

The activity stream also needs a strict disclosure boundary. Low-level does not mean raw. Event details may contain stable identifiers, structured error classes, durations, counters, and links to bounded logs. They must not contain provider secrets, reusable credentials, raw prompts, or unbounded stdout. The stream is an index into evidence, not a second log archive.

This makes the activity page more than a visualization. It becomes the place where durable intent and disposable execution meet without being confused. A `deferred` transition is a high-level event because it changes portfolio intent. A MicroVM heartbeat is low-level because it describes one disposable runtime. Both remain auditable, but they compete for attention differently.

E.7.7. What Has Landed, and What Remains

Area	Evidence at this checkpoint	Required before wave exit
current-state rollup	`yesod factory status` consolidates major operational surfaces (`ys-yes-q0e7`)	link status facts to the events that produced them
catalog activity	the `yesod serve` landing page records catalog creation and mutation activity (`ys-yes-zsy3`)	normalize catalog events with factory events instead of maintaining a separate story

Area	Evidence at this checkpoint	Required before wave exit
live page	`/viz/live-events.html` exists and consumes snapshot plus streamed Bead events (`ys-yes-u150`)	replace prototype controls and narrow normalization with the leveled, filterable operator experience
reconnect safety	cursor/backfill and snapshot recovery infrastructure exists (`ys-yes-ih57`)	prove initial history plus reconnect has no silent gaps or duplicate rendering across the unified envelope
parked AWS work	four Phase 1 features have grounded three-child trees and explicit cross-feature ordering	represent them as `deferred` rather than temporary `wontfix`, then revalidate rather than replan when revived
lifecycle vocabulary	`wontfix`, holds, dependencies, and planning each have distinct existing meanings	implement `deferred` across constraints, writers, filters, views, exports, search, distillation, and dispatch exclusion (`ys-yes-b3iy`)

E.7.8. The New Exit Criterion

Earlier waves proved that the factory could carry work, survive failure, merge through a remote gate, and stop when no action remained. Wave 7 is complete when the operator no longer needs subsystem archaeology to understand those capabilities in motion.

From one page, the operator must be able to answer:

- What happened, in chronological order?
- Which changes matter at a high, medium, or diagnostic level?
- Which tools and work objects were involved?
- What is running, waiting, completed, or intentionally parked?
- Who or what caused each transition?
- Did the stream reconnect cleanly, or is some history unavailable?

At the same time, the lifecycle must tell the truth about intent. Accepted but parked work must be `deferred`, excluded from every execution path, and recoverable without losing its grounded plan. Rejected work must remain `wontfix`. Active work must not become eligible merely because it is detailed.

The durable lesson is that observability is not the accumulation of status pages. A factory becomes legible when its state changes form a coherent, filterable narrative and when its lifecycle vocabulary distinguishes what the system can execute from what the human has authorized it to pursue.

F Gas Town, Gas City, and Yesod

Yesod does not exist in isolation. Two closely related open-source projects—**Gas Town** and **Gas City**—attack many of the same problems: durable work outside an agent's context window, parallel coding sessions, role specialization, worktree isolation, recovery after session death, and coordination across repositories.

The names can obscure the relationship. Gas Town is the older, opinionated multi-agent workspace manager. Gas City extracts its reusable machinery into a role-agnostic orchestration SDK. A real `gastown` pack then recreates Gas Town's familiar organization on top of Gas City using configuration, prompts, formulas, orders, and scripts.

This appendix compares three product shapes rather than treating the names as interchangeable:

1. the standalone **Gas Town** system and its `gt` CLI;
2. the role-agnostic **Gas City** engine and its `gc` CLI;
3. the **Gastown pack** running on Gas City.

F.1. Research Snapshot

These projects move quickly, and their prose sometimes describes a design before the default execution path has caught up. The comparison therefore uses pinned source snapshots and treats code, current configuration, and executable pack definitions as stronger evidence than older design documents.

System	Snapshot	Date / release context
Yesod	`cf091d2e`	book snapshot, 2026-07-11
Gas City	`6c8dfdce`	`main`, 2026-07-13; stable `v1.3.4`
Gas Town	`f97ed211`	`main`, 2026-07-13; stable `v1.2.1`
Gas City packs	`04fc38f6`	`gastown` pack, 2026-07-13

Gas City is the strategic successor architecture, but Gas Town is not an abandoned repository or a completed in-place migration. Both remain active. The official command map warns that their architectures are not identical, and moving from `gt` to `gc` means re-expressing the operating model rather than upgrading one workspace database in place.

> **Terminology Trap — Gas City is not merely Gas Town 2.** Gas Town is a particular organization. Gas City is a kit for constructing organizations. The Gastown pack is one organization built with that kit.

F.2. The Central Distinction

The most useful comparison is not "which system has more agents?" It is **where each system draws the boundary between judgment and mechanism**.

Yesod is **vertically deterministic**. It codifies one delivery pipeline deeply: claim rules, worktree creation, process governance, proof of work, branch publication, merge selection, gate execution, push verification, landed-commit proof, and post-merge hooks. Roles remain flexible at the planning and diagnosis edges, but the path by which code reaches `main` is deliberately narrow.

Gas City is **horizontally deterministic**. It codifies reusable orchestration machinery: desired-state reconciliation, sessions, runtime providers, elastic pools, formula graphs, dependency gates, fan-out, retries, orders, events, and health patrol. It intentionally does not prescribe a universal planner, reviewer, merge authority, or delivery policy.

Gas Town is more **agent-centric**. Its daemon and CLI provide substantial mechanics, but Mayor, Deacon, Witness, Refinery, Polecat, and Dog sessions interpret long role guides and workflow formulas to run much of the organization.

Dimension	Yesod	Gas Town	Gas City
Product posture	opinionated autonomous factory	opinionated multi-agent workspace	orchestration-builder SDK
Roles	explicit primers plus fixed services	built-in named organization	zero roles in the core
Workflow	fixed evidence pipeline plus Beads DAG	convoys, molecules, role patrols	pack-defined formula graphs
Integration	deterministic AME	agent-run Refinery	pack-defined; no core policy
Extensibility	tools, processes, profiles, lanes	role and runtime configuration	packs, providers, formulas, orders
Primary state	PostgreSQL + Beads/Dolt + Git	town/rig Beads/Dolt + Git	pluggable Beads store + events

This is not a maturity ranking. A general SDK must leave choices open that a vertically integrated factory can make mandatory.

F.3. Where the Roles Live

Gas Town makes its organizational metaphor part of the product. The Mayor is the human-facing coordinator. Deacon patrols the town. Witness watches a rig. Refinery processes its merge queue. Polecats implement bounded assignments. Dogs perform maintenance and recovery. The directory tree, role identity, tmux sessions, and operating procedures reinforce the taxonomy.

Yesod also has named roles, but it separates them from the services that enforce factory mechanics. `yesod prime --planner` or `yesod prime --triage` loads an operating contract into a session. The runner, AME, gate proxy, and other services do not become those roles; they remain code with narrower state transitions.

Gas City deliberately hardcodes no role names. An agent is a prompt, provider, scope, work query, and session policy declared by a pack. A "planner" has no privileged meaning to the Go controller. The same engine can load a Gastown pack, a Ralph-style loop, a review swarm, or a project-specific organization.

The `gastown` pack proves that this is more than an aspiration. It defines Mayor, Deacon, Boot, Dog, Witness, Refinery, and Polecat agents; patrol and shutdown formulas; scripts; orders; commands; and tests. Crew and polecat become operating styles rather than platform types. A Gas City feature in the controller can therefore improve every pack without teaching the core what a Witness is.

> **Design Trade-off — Law versus configuration.** A hardcoded role system is easier to understand but harder to recombine. A role-free substrate is easier to extend but cannot, by itself, guarantee that a pack chose safe responsibilities.

F.4. Planning and Decomposition

All three systems put ambiguous planning primarily in agents.

In Yesod, a planner reads source and chooses merge boundaries. Each feature gets one note and root Bead; child Beads form its internal checklist, and note dependencies order separately mergeable features. The planner rates complexity and leaves the notes unarmed. The durable plan survives the session, and deterministic services later decide readiness and claimability after the dispatcher chooses lanes.

In Gas Town, the Mayor commonly turns intent into issues, convoys, formulas, and Polecat assignments. Planning conventions are supplied through the Mayor's role guide and workflow definitions.

In Gas City, planning is pack policy. A formula can encode a repeatable planning method, including parallel review legs and human gates, but an agent usually performs the semantic decomposition. Once a version-2 formula has materialized a graph, the controller—not the originating agent—drives its dependency and control structure.

That last distinction is important. Gas City formulas can contain deterministic **control beads** for checks, retries, fan-out, drains, tallies, and finalization. Agents execute work beads; the controller advances the graph outside any one session. This is more general than Yesod's

current fixed note-to-Bead dispatch path, although Yesod's path has stronger delivery-specific proof.

F.5. Claim, Execution, and Isolation

Yesod's runner polls for the intersection of an eligible note and a dependency-ready Bead, wins ownership through a short PostgreSQL transaction, assigns the Bead, creates a new Git worktree, launches the selected lane, enforces time and token budgets, and independently reconciles the result. The agent is not trusted to decide whether its own transcript proves progress.

Gas Town uses Beads hooks, sling operations, a capacity scheduler, and Polecat sessions. Work is commonly isolated in a Polecat worktree. `gt done` performs substantial mechanical completion work, but the larger workflow still assumes the Polecat follows its injected formula and completion discipline.

Gas City separates these concerns into providers and configuration. A controller reconciles desired agents to sessions, sizes pools from demand queries, and can run through tmux, subprocesses, ACP, Kubernetes, SSH, `exec:` providers, or other runtime packs. Worktree isolation is available through pack startup hooks and `work_dir`, but it is a pack choice rather than a universal platform invariant.

Gas City is consequently stronger as a reusable runtime layer. Yesod is stronger at making one execution contract mandatory for factory-dispatched code work.

F.6. Integration Is the Clearest Divergence

The sharpest difference is what happens after a worker says it is finished.

In Yesod, the runner proves that qualifying commits exist, executes declared acceptance checks, pushes a named branch, and verifies the remote branch. The AME then resets a dedicated checkout, verifies candidate branches, chooses a compatible batch, performs the merge, runs the authoritative gate, pushes `main`, verifies the remote SHA and commit ancestry, records the result, and invokes the configured post-merge hook.

The worker cannot turn its own assertion into a merge. The merge process does not need an LLM to remember the sequence.

Gas Town has a Go package containing deterministic merge primitives, claims, gates, and batch/bisect code. Its design documentation describes a Bors-style queue. At the inspected snapshot, however, the normal Refinery manager still starts an LLM agent in tmux, and the batch `ProcessBatch` path has no non-test caller. The current Refinery guide instructs the agent to select work, run Git commands, classify failures, decide whether a conflict is trivial, run tests, and push.

The Gastown pack on Gas City states the boundary even more directly: merge and conflict decisions belong to the Refinery agent, not to Go code. Its patrol formula contains the rebase,

test, rejection, branch cleanup, merge, push, and Bead-update procedures that the agent is expected to interpret and execute.

Gas City core does not define a universal merge queue because a generic orchestration SDK cannot assume every workflow produces Git branches. A pack can implement a deterministic executable order, an agent-driven Refinery, a pull-request workflow, or no code integration at all.

> **Invariant — Preserve the narrow door.** A Yesod integration with Gas City should use Gas City's runtime and graph machinery without moving AME authority into a pack prompt. The pack may request integration; deterministic Yesod services should continue to prove and perform it.

F.7. Recovery and Health

Gas Town's recovery system mixes code and agent judgment. The daemon monitors mechanical liveness and restarts important sessions. Deacon assesses town-level progress. Witness investigates rig work, orphaned Beads, and stuck Polecats. Dogs execute shutdown warrants. This permits nuanced recovery, but it also spends model capacity on operational loops and depends on agents correctly following long recipes.

Gas City moves more of that infrastructure into its controller. Health Patrol performs desired-state reconciliation, configuration fingerprinting, idle retirement, restart-window accounting, crash-loop quarantine, pool draining, orphan cleanup, bounded dependency-aware starts and stops, order dispatch, and wisp garbage collection without requiring a Deacon role.

The Gastown pack can still add Deacon, Witness, and Dog agents for work-layer questions: Is work actually progressing? Is a failure structural or transient? Is an apparently silent process performing a legitimate long tool call? The architectural improvement is that these agents advise above a deterministic supervision substrate rather than owning basic process existence.

Yesod sits between these models. Its runner and AME perform strong deterministic reconciliation around execution and integration. Its SPS and watchdog layers are less mature than Gas City's general session supervisor, especially around chronic degradation, cause-aware recovery, and cross-host actions. Gas City's crash quarantine, content-hash drift detection, provider abstraction, and bounded lifecycle waves are directly reusable ideas.

F.8. Parallelism

Gas Town parallelizes by slinging independent Beads to Polecats, potentially across rigs. A scheduler can cap concurrent Polecats so a large convoy does not exhaust API quota, memory, or CPU. Its default operational center remains a local town with tmux-backed sessions.

Gas City elevates parallelism into the formula engine. Independent graph steps become ready together, elastic pools expand to demand, review lanes fan out, drains wait for whole runtime-discovered sets, and the controller advances the graph outside the human's session. Runtime providers also make the execution location replaceable.

Yesod parallelizes ready Beads across eligible runner hosts and model lanes. It adds atomic claims, per-run worktrees, host affinity, and a separate integration queue. Its parallelism is currently less general than Gas City's arbitrary workflow graph, but more tightly coupled to verified code delivery.

The strategic difference is the unit of reuse:

- Gas Town reuses an **operating model**;
- Gas City reuses an **orchestration language and runtime**;
- Yesod reuses an **evidence-governed factory path** across tools.

F.9. Durable State and Evidence

Gas Town uses a two-level Beads architecture. Town-level Beads hold cross-rig coordination, identity, convoys, and mail. Rig-level Beads hold implementation issues, merge requests, and workflow state. A shared Dolt SQL server provides versioned storage; Git branches and worktrees hold code.

Gas City makes Beads the universal work substrate. Tasks, sessions, messages, convoys, formula runs, and control state are different typed Beads with labels and metadata. Append-only sequenced events provide observation and replay. The default provider uses `bd` and Dolt, while file and executable providers keep the store boundary replaceable.

Yesod deliberately keeps three truths separate. PostgreSQL holds narrative intent, dispatch, runs, merge jobs, and telemetry. Beads/Dolt holds dependency structure. Git holds source ancestry. This costs more reconciliation code, but gives operationally distinct questions distinct records.

> **Design Trade-off — One substrate or several truths.** Gas City's universal Bead model simplifies composition. Yesod's split model makes the difference between intent, work readiness, execution evidence, and landed code harder to blur.

F.10. Cost and Model Selection

Gas Town supports multiple coding harnesses and static cost tiers. An operator can use cheaper models for patrol roles and stronger models for workers. Cost commands estimate retrospective session spend, but dynamic task-level routing remains more design than default operational policy.

Gas City separates harness, model, upstream provider, transport, and runtime into independent configuration axes. Current `main` records model tokens, compute time, and list-price estimates as usage facts and exposes `gc costs`. Model selection nevertheless remains primarily static per configured agent or pool. Cost observations do not yet close the loop into dispatch.

Yesod is more opinionated about the economic unit. A dispatch decision stores its lane and reason; the run ledger preserves every attempt; statistics correlate cost with merges and churn. The target metric is cost per accepted change, not price per token. Yesod's router

is still not a fully autonomous optimizer, but its data model is closer to learning which lane delivers value for a particular project and task class.

F.11. Knowledge Beyond Work Tracking

Gas Town and Gas City center orchestration. They are rich in tasks, formulas, sessions, mail, runtime state, and reusable configuration.

Yesod also treats software tools as a knowledge ecology. Tools have durable notes, feature requests, bug reports, processes, calls, topics, and cross-tool relationships. An agent can ask which tools exist, how they compose, what failed before, and which process should be poured. Tools can accumulate requests against one another.

Gas City's pack registry and Gas Town's federation ideas are adjacent, but they are not the same abstraction. A pack answers "how should this organization run?" The Yesod catalog answers "what tools and processes does this organization know, and what have they learned about one another?"

F.12. Authority and Trust

All three systems assume substantial local trust, but the boundaries differ.

Gas Town agents normally run as the user, share filesystem and network access, and can reach configured credentials and remotes. Role identity is conveyed through prompts, environment, directories, and Beads attribution rather than an unforgeable capability system. Its security documentation is explicit that stronger Polecat sandboxing remains future work.

Gas City improves the abstraction boundary and documents it carefully, but is intentionally not a sandbox. City configuration, imported packs, provider scripts, startup commands, and executable orders are trusted code. It strips ambient secret-looking environment variables from orchestrator-side helpers and offers more explicit provider and grant surfaces, but a privileged imported pack must still be reviewed like executable code.

Yesod's peer identity is also self-asserted, and role priming is not a credential. Its stronger property is a data-flow rule: peer mail carries no human authority, workers do not merge, and deterministic integration services own the normal automated path to `main`. That architectural separation reduces the number of agent sessions that require integration authority, although repository hosting must still enforce the desired credential policy if an absolute boundary is required.

F.13. What Yesod Should Borrow

Gas City contains several ideas that would strengthen Yesod without changing its identity:

1. **A runtime-provider interface.** Separate harness, model, upstream, transport, and execution location instead of encoding each lane as a largely bespoke command.

2. **Declarative packs.** Package role primers, processes, health policies, runtime presets, and tool-specific defaults as versioned imports.
3. **Controller-grade health patrol.** Add configuration fingerprints, crash-loop quarantine, provider-neutral liveness, bounded start/stop waves, and explicit recovery states.
4. **Formula control nodes.** Make fan-out, retry, drain, tally, and finalization first-class deterministic process operations.
5. **A sequenced event bus.** Preserve durable tables as truth while emitting a replayable, typed operational stream.
6. **Clear trusted-code documentation.** Treat imported workflow definitions and executable hooks as dependencies with explicit review and pinning rules.

F.14. What Yesod Should Keep

The comparison also clarifies what should not be generalized away:

1. **The deterministic AME.** Do not turn merge ownership back into an agent role that remembers Git procedures from a prompt.
2. **Independent proof of worker output.** Preserve commit, branch, acceptance, push, and ancestry checks outside the coding process.
3. **The run ledger.** Keep attempts, failures, tokens, cost, verification, and merge attribution as durable operational evidence.
4. **The catalog and process ecology.** Tool knowledge and reciprocal improvement are a different product advantage from orchestration packs.
5. **Delivered-outcome economics.** Optimize toward accepted changes rather than configured model price.
6. **The human-authority channel.** Keep authority separate from peer coordination and claimed role identity.

F.15. A Practical Integration Path

Gas City is most interesting to Yesod as a **session and workflow substrate**, not as a replacement for the factory's evidence core.

A future `yesod` pack could define the Mayor, planner, dispatcher, triage, infra-monitor, and experimental roles; provide their prompts and runtime defaults; and use Gas City pools and providers to keep those sessions alive. Version-2 formulas could express planning reviews, diagnostic swarms, documentation audits, and other workflows whose control structure benefits from deterministic fan-out and drain.

The pack would call Yesod's existing catalog and Beads-routing commands. Factory-dispatched code work would still enter the Yesod note/run ledger. The runner would still prove the branch. The AME would still own integration and deployment proof.

That composition preserves the best boundary in each design:

> **Gas City manages how agent sessions exist and cooperate; Yesod decides what constitutes proved software delivery.**

F.16. Sources

Official sources inspected for this appendix:

- Gas City README
- How Gas City Works
- Coming from Gas Town
- Gas City Health Patrol
- Gas City trust boundaries
- Gas City `gastown` pack
- Gastown pack Refinery prompt
- Gas Town README
- Gas Town architecture
- Gas Town Refinery manager
- Gas Town Refinery role
- Gas Town provider integration

G Glossary

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

AME — Autonomous Merge Engine: supervisor plus one-shot ephemeral merge workers.

Armed — a note whose `agent_dispatch` holds a real lane. Not a lifecycle status.

Bead — a work item in a per-tool Beads/Dolt graph.

Bead dependency — ordering between checklist items inside one feature's Bead tree; not a merge boundary.

Blocked — work stopped by a retry/no-progress/integration cap and needing changed conditions or judgment.

Catalog — PostgreSQL-backed registry of tools, notes, processes, runs, queue state, and telemetry.

Control plane — agents that make planning, routing, and diagnosis judgments.

Data task — a task whose acceptance probe, rather than Git commits, proves completion.

Dispatch — assigning work to an execution lane and allowing eligible runners to claim it.

Dispatcher agent — role that chooses and arms lanes; not the runner daemon.

Dolt — versioned SQL storage used by Beads workspaces.

Ephemeral merge worker — one-shot AME subprocess that performs one integration attempt.

ERS — judgmental ephemeral-refinery-supervisor agent; not the AME daemon.

Gate — must be qualified. The merge gate tests a candidate tree; Beads approval gates control work decisions.

Gate-on-merge — downstream work stays blocked until the prerequisite note lands or is superseded, even if its bead closed early.

Governing note — the feature note linked to a root Bead and armed for the checklist tree beneath it. The runner retains older exact-child compatibility, but planners no longer create child notes for ordinary checklist Beads.

Hold — explicit `agent_dispatch = hold`, removing a note from ordinary claims.

Lane — named model/backend target; not a host.

Merge job — durable record of one AME integration attempt.

Merge queue — durable but transient working set for branches awaiting integration.

Merge summary — durable record that a candidate landed.

Molecule — concrete Beads graph poured from a reusable process prototype.

Note — durable tool-attached knowledge or work record in PostgreSQL.

Note dependency — merge-aware ordering between independently mergeable feature notes; satisfied only when every prerequisite is `done` or `superseded`.

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

Refinery profile — per-tool gate, repository, branch, deploy, and ownership configuration.

Runner daemon — `yesod codefactory-dispatch`; deterministic execution scheduler.

Run ledger — one durable row per dispatched execution attempt.

Service — deterministic long-running or one-shot mechanism.

Tool — catalog entity representing software the user owns, shares, or depends on.

Topic — durable cross-cutting knowledge object not owned by one tool.

Worker agent — coding process launched by a runner inside a worktree.

Worktree — isolated Git checkout for one execution or merge context.

H Building This Book

The book is deliberately reproducible with a small local toolchain:

- Pandoc 3.10 or compatible;
- XeLaTeX;
- Python 3 (standard library only) for deterministic SVG diagrams;
- `rsvg-convert`;
- the Charter, Avenir Next, and Menlo fonts on macOS.

From the `book` directory:

```
make
```

The build performs three steps:

1. render the deterministic SVG diagrams;
2. convert the diagrams to vector PDF figures;
3. run Pandoc with the custom KOMA-Script/XeLaTeX theme.

The exact Pandoc invocation and canonical source order are preserved in `book/Makefile`. The manuscript is split across `book/frontmatter/`, `book/chapters/`, and `book/appendices/`; `book/yesod-software-factory.md` is a human-readable source map rather than the manuscript body. Development waves live one-per-file under `book/appendices/development-waves/`. Metadata and geometry live in `book/metadata.yaml`; typography and the title page live in `book/tex/preamble.tex`; editable diagrams are generated by `book/figures/render.py`.

Useful validation:

```
make check
pdfinfo yesod-software-factory.pdf
pdftotext yesod-software-factory.pdf - | wc -w
```

H.1. Publishing an edition

The `Publish book edition` GitHub Actions workflow rebuilds and validates the book on macOS whenever canonical book sources change on `main`, and can also be started manually. It calls `scripts/publish-book.sh`, which:

1. chooses an unused immutable dated key in the `yesod-publications` R2 bucket;
2. uploads the PDF with immutable cache and download metadata;

3. downloads it again and verifies its byte count and SHA-256 digest; and
4. sends a `book-edition-published` repository dispatch to `yesod.work`.

If multiple editions are published on one UTC date, later objects receive `-r2`, `-r3`, and so on. Existing objects are never overwritten. The website receiver opens a manifest pull request; publication is not visible on `yesod.work` until that pull request is reviewed and merged.

For a portable edition, replace the macOS fonts with installed open fonts in `metadata.yaml`. The content does not depend on a proprietary template or network resource.