OPERATING MODEL & BUILD GUIDE

A multi-agent system of specialists, contracts, and quality gates

How to run product and software-delivery work through a coordinator and a set of specialist AI agents — each one both an expert and a quality gate — and how to actually build it using custom agents, shared steering, specs, and hooks.

Author	Date	Scope
Mickey Bushman, Principal Product Manager	2026	Software delivery — Coordinator + eight specialists

1 Overview

An effective multi-agent system is built on three things: clear authority boundaries, immutable contracts between agents, and verification at every handoff. The organising idea is that **agents own outcomes, not implementations**.

An upstream agent defines what must be accomplished — constraints and acceptance criteria. The receiving agent decides how, within its own domain expertise, and is accountable for the result. Every agent is therefore both a specialist (it produces one kind of artifact) and a quality gate (it independently validates what it receives before accepting it). The payoff is less rework, no domain overreach, and a system that scales cleanly as you add specialists.

> **One sentence.** A coordinator routes work; specialists produce and own artifacts; every handoff carries a fixed contract and is independently verified; corrections always return to the producing owner; and ownership never changes hands.

2 Operating principles

Every agent — coordinator and specialists alike — follows the same rules.

- **Accept objectives, not implementation instructions.** An agent takes constraints and acceptance criteria. It rejects instructions that tell it how to do its specialty — the receiving agent is the expert.
- **Never trust previous agents.** Every deliverable is independently validated on receipt. Verification is mandatory, not optional.
- **Never modify another agent's work.** If something is wrong, an agent requests a correction; it does not edit the other agent's artifact.
- **Every artifact has exactly one owner.** Product roadmap & priorities → Product Manager; Requirements → Business Analyst; Schedule & plan → Project Manager; UX → UX Expert;

Architecture & ADRs → Solution Architect; Code → Development; Tests → QA; Documentation → Technical Writer.

- **No agent skips validation.** Nothing advances without passing the next responsible agent's checks.
- **Right-size the intelligence.** Each agent runs on the smallest model sufficient to perform its role at a high level — and nothing more. A routing coordinator does not need frontier-model reasoning; an architect weighing non-functional trade-offs does. Matching model capability to role keeps cost proportionate and latency low, and reduces overreach: an over-powered model given a narrow job is the one most tempted to work outside its lane.

> **What “reject” means here.** Rejecting isn't refusal — it's returning work to its owner with a structured reason (missing input, conflict, mis-owned content, or an authority breach). The four checks in section 6 define exactly when to reject.

3 The agent roster

One coordinator who only routes, and eight specialists who each own one artifact and act as a gate on the work they receive.

Coordination Agent	*Executive Orchestrator — owns workflow & routing; no specialist work*
MANAGES	Workflow, dependencies, priorities, sequencing, escalation, conflict resolution, lifecycle state.
ASKS	Is enough information available? Which specialist owns the next step? Is the project blocked? Has every artifact passed validation?
DOES	Start projects, route work, detect missing artifacts, handle failed reviews, decide sequencing, close completed work.
NEVER	Evaluates code, UX, or requirements quality — those belong to specialists. It routes; it does not judge content.

Product Manager	*Owns product direction — the why, the what, the priority*
OWNS	Product vision & strategy for the initiative, the problem/opportunity definition, target customers and value proposition, the prioritised roadmap, scope (in/out) decisions, and outcome success metrics (business KPIs).
INPUTS	Business goals, market and customer signals, and constraints from the Coordinator / stakeholders.
OUTPUTS	A product brief / opportunity definition, prioritised scope, and success metrics — the objective the Business Analyst turns into detailed requirements.
VALIDATES	Before accepting an objective: is the business goal and desired outcome clear? Returns to the Coordinator / stakeholders if the “why” is missing.
REJECTS	A pre-baked feature spec or solution (“build screen X”) — reframes it to the underlying problem and the outcome to achieve; never specifies implementation.
SEAM	PM decides which problems to solve and why (value & priority); the Business Analyst defines the detailed what; the Project Manager plans when / how.
TOOLS	Read all artifacts (for value/scope alignment); write only to product/ (brief, roadmap, metrics); no requirements, code, or schedule.

The human/PM-agent seam. The PM agent drafts briefs, guards scope, and enforces prioritisation discipline — but it does not decide what to build. Customer judgment, prioritisation calls, and scope trade-offs belong to the human product manager; the agent operationalises those decisions. This is the one seam in the system that is deliberately not delegable.

Product Manager ≠ Project Manager. The Product Manager owns the product — what to build, for whom, and in what priority to maximise value (roadmap, outcomes). The Project Manager owns execution — turning approved work into a backlog, schedule, and dependencies. Two owners, two artifacts; the value chain runs Product Manager → Business Analyst → Project Manager.

Business Analyst	*Owns problem definition & requirements*
OWNS	Requirements, business rules, assumptions, constraints, acceptance criteria, personas, success metrics.
INPUTS	A business request / objective from the Coordinator.
OUTPUTS	A complete specification: functional + non-functional requirements and testable acceptance criteria.
REJECTS	Implementation guidance. “Add login” becomes a full spec; it never says JWT, OAuth, cookies, or which database — those belong to Development.

Project Manager	*Owns execution planning*
OWNS	Milestones, work breakdown, backlog, priorities, estimates, dependencies, sprint plan.
INPUTS	Approved requirements.
OUTPUTS	Epics, stories, tasks, a dependency graph, a delivery schedule.
VALIDATES	Before accepting requirements: scope, priority, and business value are defined. If not, returns to the BA.
NEVER	Writes requirements, code, or UX; deciding “split into 8 tasks” is the PM's call, not an instruction it accepts.

UX Expert	*Owns user experience*
OWNS	User journeys, interaction models, workflows, layout, accessibility, usability, visual hierarchy.
INPUTS	Approved business requirements.
OUTPUTS	Wireframes, interaction specs, accessibility notes, design tokens, workflow diagrams.
REJECTS	“Put Save in the upper right.” It receives “users must easily save progress” and decides the mechanism (autosave, sticky footer, …).
VALIDATES	Every requirement represented; accessibility; usability; consistency.

Solution / Chief Architect	*Owns system architecture & technical constraints*
OWNS	System and solution design, architecture decision records (ADRs), integration and interface boundaries, data architecture, and the system-level non-functional budgets (scalability, availability, security posture, performance).
INPUTS	Approved requirements, approved UX, and business/technical constraints from the Coordinator.
OUTPUTS	The solution architecture, ADRs, integration/interface contracts, and the technical constraints Development consumes as inputs.
AUTHORITY	Sets system-level constraints and boundaries — not Development's internal choices. “Encrypt PII at rest” and “stateless behind the gateway” are architecture; which library or pattern realises them is Development's.
VALIDATES	Before accepting requirements: are the quality attributes / NFRs defined enough to architect against? Reviews UX for system-level feasibility; returns to BA/UX if not.
REJECTS	An architecture pre-decided by the business with no driving requirement (“use microservices”); requests to write code; requirements that smuggle in implementation.
SEAM W/ DEV	Architect owns the what / where / boundaries (ADRs); Development owns the how (code + component design) within those ADRs, and may reject anything that crosses into implementation.
TOOLS	Read requirements/UX/code; write only to architecture/ (ADRs & design); no code.

Development	*Owns implementation*
OWNS	Implementation within the architecture — component design, frameworks, algorithms, patterns, optimisation.
INPUTS	Approved requirements, approved UX, approved architecture (ADRs), technical constraints.
OUTPUTS	Source code, component/implementation docs, APIs, migrations.
REJECTS	“Use React hooks / store in Redis / make three services” — unless a stated architecture constraint requires it.
VALIDATES	Before submitting: build, lint, static analysis, unit tests, documentation all pass.

QA	*Owns quality verification*
OWNS	Test planning, regression, edge/negative cases, exploratory testing, automation, performance verification.
INPUTS	Requirements, UX, code, acceptance criteria.
OUTPUTS	Pass / Fail, defects with severity and reproduction steps.
RULE	Compares implementation to requirements, not to developer intent. Never fixes code, never changes requirements — files defects back to Development.

Technical Writer	*Owns documentation*
OWNS	User guides, reference/API docs, release notes, runbooks, READMEs, in-product help, changelogs.
INPUTS	Approved requirements, approved UX, shipped + approved code/architecture, the QA pass with acceptance criteria.
OUTPUTS	The documentation set, in the house style, with the right voice per audience.
VALIDATES	Accuracy against shipped behaviour (not intent or stale specs); completeness (every user-facing task covered); consistency (terminology, product name, style); doc accessibility; confidentiality clearance.
REJECTS	Prescribed wording/structure beyond constraints; requests to document unshipped or unverified behaviour (flags the gap instead); content owned elsewhere.
INHERITS	The documentation steering — style, types, personas, confidentiality — plus the render-and-QA loop for visual docs.

4 Workflow & review loops

The spine is a pipeline, but it is not strictly linear — each handoff is gated, and downstream specialists may review upstream artifacts early (advisory only).

```
Coordinator
     ↓
Product Manager        (why / what / priority)
     ↓
Business Analyst       (requirements)
     ↓
Project Manager        (plan / backlog)
     ↓
UX Expert              (experience)
     ↓
Solution Architect     (architecture / constraints)
     ↓
Development            (implementation)
     ↓
QA                     (verification)
     ↓
Technical Writer       (documentation)
     ↓
Coordinator            (route / close)
```

Cross-cutting advisory review loops

Before a deliverable advances, relevant downstream specialists may review it early without modifying it. Reviews are advisory; only the owner changes the artifact, and unresolved conflicts go to the Coordinator.

- **Development** reviews requirements for technical feasibility and raises questions to the BA — it does not rewrite them.

- **QA** reviews requirements and UX early to catch ambiguous acceptance criteria before implementation begins.
- **Solution Architect** reviews requirements and UX early for technical feasibility and non-functional implications, and sets the constraints Development will build within — without writing code.
- **UX** reviews requirements for usability risks and requests clarification where user goals are unclear.
- **Technical Writer** reviews requirements and UX early — "is this documentable, are terms defined, do we need a glossary?" — and flags gaps back to the owner.
- **Project Manager** continuously checks that scope, estimates, and dependencies stay aligned as artifacts evolve.
- **Product Manager** guards value and scope — confirms each evolving artifact still serves the prioritised outcome, and rules on scope trade-offs (escalating to the Coordinator only when priorities truly conflict).

5 The handoff contract

Every work item passed between agents carries the same fixed structure — and nothing else. There are no implementation instructions.

```
# Handoff Contract
Owner:               <the agent accountable for the resulting artifact>
Objective:           <what must be accomplished — an outcome, not a method>
Inputs:              <approved upstream artifacts this work depends on>
Constraints:         <hard limits: security, compliance, platform, budget>
Deliverables:        <the artifact(s) to be produced>
Acceptance Criteria: <testable conditions that define "done">
Open Questions:      <anything unresolved the owner must close or escalate>
Dependencies:        <other artifacts/agents this blocks or is blocked by>
Validation Checklist: <the checks the receiver runs before accepting>
```

> **Example — Product Manager → Business Analyst.** Objective: "Create complete requirements for customer authentication." Acceptance: requirements support implementation; business rules documented; security constraints defined. No technical implementation guidance. The BA decides the rest. Likewise BA → Project Manager never says "split into 8 tasks"; UX → Solution Architect → Development flows as requirements and constraints, never a named framework. No contract ever tells the receiver how to do its job.

6 Validation, corrections & loop control

Every receiving agent runs four checks before accepting work. Any failure returns the item to its owner — and corrections are bounded so that work always converges rather than ping-ponging.

Check	Question	Reject when…
1. Completeness	Is all required information present?	An input, constraint, or acceptance criterion is missing.

2. Consistency	Does anything conflict?	Requirements contradict UX, or two inputs disagree.
3. Ownership	Is this work owned elsewhere?	A BA spec includes a SQL schema (that's Development's).
4. Authority	Does the sender tell the receiver how?	“Use PostgreSQL” → rejected; becomes “persist user data reliably.”

Correction cycle

Corrections always return to the producing agent — ownership never moves. If QA finds a defect, QA does not edit code; it files the defect, Development fixes it, QA retests. Same for every artifact.

Escalation

The Coordinator intervenes only on: a deadlock, an ownership dispute between two agents, conflicting priorities, requirements that cannot be satisfied simultaneously, project risk above threshold, or a timeline change. Otherwise, specialists resolve their own work.

Termination & loop control

Corrections must converge, not ping-pong. Three rules guarantee an item always ends:

- **Iteration caps.** A work item may cycle between the same two agents at most twice; on the third failure the Coordinator must intervene, and once a small total budget is spent (e.g. five cycles across an item) it escalates to a human. Caps are what actually break loops.
- **Acceptance is criteria-bound, not taste-bound.** An artifact passes the moment it meets its stated acceptance criteria. A reviewer cannot withhold approval for preferences outside the criteria, and cannot introduce new scope during review — that kills the perfectionism loop. (It is QA's “compare to requirements, not intent,” generalised to every reviewer.)
- **Advisory reviews converge.** Advisory feedback is timeboxed and non-blocking; the same point cannot be raised twice — if a reviewer still disagrees after the owner responds, it goes to the Coordinator, not into another round.

Ping-pong & deadlock detection. The Coordinator watches STATE.md for an item rejected back and forth, or an ownership dispute between two agents, and makes a binding routing/ownership call rather than letting the bounce continue.

Change-control on baselines. Re-opening a baselined artifact is a deliberate, owned revision with its own cap — not an open-ended re-edit by whoever noticed the issue.

The human circuit-breaker

The Coordinator is an agent, so it cannot be the final backstop. A named human is the escalation endpoint and the approver for anything irreversible or high-risk. The Coordinator surfaces to the human when an iteration cap is hit, the same conflict escalates twice, project risk exceeds threshold, or an action is irreversible (production change, data deletion, external commitment). Nothing high-stakes resolves purely between agents — this is also the real enforcement behind the “convention, not guarantee” caveat in section 9.

Keeping work moving (throughput)

- **Safe parallelism.** Work that depends only on baselined inputs can run concurrently — e.g. the Technical Writer drafts from baselined requirements/UX while QA runs; UX and the Solution Architect overlap once requirements are baselined. Never parallelise on a not-yet-baselined artifact — that is what causes rework.
- **WIP limits.** The Coordinator starts no more items than the pipeline can finish; an item must reach a gate before the next is pulled in. Limiting work-in-progress is the simplest throughput lever.
- **Efficiency signals.** STATE.md tracks cycle time, rework rate, rejection count, and first-pass yield per item, so stalls and loops are visible — not discovered at the end.

Failure handling (distinct from rejection)

A rejection is a valid outcome routed to an owner. A failure — a hook command that won't run, an agent that errors, a missing tool — is different: retry once, and if it still fails the Coordinator records it in STATE.md and escalates to the human. A failure never silently drops an item.

7 Definition of ready & done

Definition of Ready (entry gate)

Most loops start because work entered underspecified. Before an agent accepts an item it must pass the four checks and a Definition of Ready: the objective is stated as an outcome; required inputs are present and baselined; constraints and acceptance criteria are explicit; dependencies are known. If it isn't ready, it is rejected on entry — far cheaper than discovering the gap three handoffs later.

Definition of Done (exit lifecycle)

Every artifact then moves through the same lifecycle, and never advances without approval from the next responsible agent.

```
Draft → Internal Validation → Submitted → Independent Review → Approved → Baselined
```

“Baselined” means the artifact is frozen as an approved input for downstream work; changing it afterward is a new, owned revision — not an edit by whoever noticed the problem.

8 Implementing the model

The model maps onto four primitives available in any modern agentic development platform (the build below was done in Kiro; the same mapping applies to any platform with custom agents, shared steering/rules, spec files, and hooks). None of it requires anything exotic — the work is in writing the agent charters and the shared steering precisely.

Model concept	Platform mechanism

Specialist agents	Custom agents — one per role, each with a charter (system prompt) encoding its ownership, authority limits, and validation checklist, plus a scoped tool set **and a right-sized model assignment** (the smallest model sufficient for the role).
Coordination Agent	An orchestrator custom agent that only routes — it invokes specialists via sub-agent delegation and performs no specialist work.
Philosophy, contract, checks	Steering files — shared, always-applied rules every agent reads: the operating principles, the handoff-contract template, the four checks, correction/escalation rules.
Requirements / plan / tasks	Specs — requirements.md / design.md / tasks.md are a ready-made backbone for the BA, PM, and execution artifacts, with verification built in.
Validation at handoffs	Hooks — e.g. postTaskExecution runs tests (QA gate); a pre-write hook can guard ownership boundaries.
Shared state / artifacts	The filesystem — an artifacts folder plus a state file the Coordinator maintains (there is no built-in lifecycle engine).

8.1 Step-by-step build

- Write the shared steering (the operating contract every agent obeys). One file, always included.
- Create each custom agent with a charter that states what it owns, what it accepts/rejects, its validation checklist, and its tool scope. Scope tools to its domain (e.g., the Technical Writer can write only under docs/). Assign each agent the smallest model that performs its role at a high level — and nothing more.
- Create the Coordinator agent whose only job is to route, track state, and escalate — never to produce or judge content.
- Set up the artifact + state layout (below) so handoffs and lifecycle are visible and durable.
- Add hook gates for the mechanical checks (build/lint/tests on the Dev→QA boundary; render-and-QA on doc deliverables).
- Pilot one small item end to end before scaling the roster.

8.2 The shared steering (excerpt)

```
---
inclusion: always
---
# Multi-Agent Operating Contract

You are one agent in a specialist pipeline. Obey these rules regardless of role:

1. Accept objectives, not methods. If a handoff tells you HOW to do your
   specialty, reject it and restate the requirement as an outcome.
2. Validate every input before accepting (Completeness, Consistency,
   Ownership, Authority). On any failure, return to the producing agent
   with a structured reason. Do not proceed.
3. Never modify another agent's artifact. File a correction request.
4. You own exactly one artifact type. Do not produce content owned by others.
5. Emit a Handoff Contract for everything you pass on (template below).
6. Record every state change in artifacts/STATE.md.

# Handoff Contract (the only inter-agent format)
Owner / Objective / Inputs / Constraints / Deliverables /
Acceptance Criteria / Open Questions / Dependencies / Validation Checklist
```

8.3 A specialist charter (example: Technical Writer)

```
# Agent: Technical Writer
Owns: documentation (guides, reference, release notes, runbooks, READMEs).
Inputs accepted: approved requirements, approved UX, shipped+approved code,
                 QA pass + acceptance criteria.
Produces: the doc set in house style, correct voice per audience.
Authority: I decide structure, format, and wording. I reject prescribed
           wording and any request to document unshipped/unverified behaviour.
Validation before submit:
  - accuracy verified against shipped behaviour (not intent / stale specs)
  - completeness (every user-facing task covered, no dead ends)
  - consistency (terminology, product name, house style)
  - doc accessibility + confidentiality clearance
Corrections: gaps/defects are filed back to BA/UX/Dev/QA; I never edit them.
Model: mid-tier — strong prose, no deep-reasoning premium.
Tools: read code/specs/UX; write ONLY under docs/; run render + layout QA.
```

Each other specialist gets the same shape of charter (owns / inputs / authority / validation / corrections / model / tools), drawn from section 3.

8.4 A hook gate (example: QA on the Dev boundary)

```
{
    "name": "QA gate on task completion",
    "version": "1.0.0",
    "when": { "type": "postTaskExecution" },
    "then": { "type": "runCommand", "command": "npm run build && npm test" }
}
```

Mechanical checks (build, lint, tests, doc render) belong in hooks so they run every time and aren't subject to an agent “forgetting.” Judgement checks stay in the agent charters.

8.5 Shared state & artifacts

```
artifacts/
    contracts/                # one handoff contract per work item
    requirements/             # owned by BA
    plan/                     # owned by PM
    ux/                       # owned by UX
    code/   (the repo)        # owned by Development
    qa/                       # owned by QA
    docs/                     # owned by Technical Writer
    STATE.md                  # Coordinator-maintained lifecycle board
```

STATE.md is the Coordinator's source of truth: per work item, its current lifecycle stage (Draft → … → Baselined), current owner, open blockers, and the loop-control signals the section 6 thresholds act on — correction-cycle count, time-in-stage / age, and rejection count. Because agents don't message each other directly, this file is the shared memory.

9 Enforced vs. convention — read this before you rely on it

The system works, but be clear-eyed about which guarantees are real mechanisms and which are behaviours held in place by the agent charters and the Coordinator's discipline.

> **Real mechanisms.** Distinct specialist agents with separate charters, scoped tools, and right-sized models; an orchestrator that routes via delegation; always-on steering that every agent obeys; specs as the artifact backbone; hooks that run mechanical checks every time; the filesystem as durable shared state.

> **Convention, not a hard guarantee.** “Never trust previous agents,” “never modify another agent's work,” “reject out-of-domain instructions,” “ownership never changes” — these are enforced by charters + steering + the Coordinator, not by sandboxing. They hold as well as the prompts are written. Where a boundary really matters, back it with a mechanism: scope an agent's write access to its own folder, and add a pre-write hook to block edits outside it.

- **No agent-to-agent messaging.** Coordination passes through the Coordinator and shared files; specialists don't talk directly mid-flight.
- **No continuous runtime.** Agents don't sit waiting on each other; the Coordinator drives the next step (hooks cover event-triggered actions).
- **Bounded context.** The Coordinator's “view of the whole” lives in STATE.md, not in memory — keep that file current or the orchestration drifts.
- **Independent verification is behavioural.** “QA compares to requirements, not intent” works because the QA charter says so and QA can read the requirements artifact — not because the platform forces a clean-room.

10 Worked example — the pilot

Prove the handoffs and gates on one small, real item before scaling. Suggested first run: a self-contained feature with a doc deliverable.

1. Coordinator opens the item in STATE.md and hands the Product Manager an objective-only contract.

2. Product Manager frames the opportunity, target outcome, success metrics, and scope/priority; baselines the brief and hands the BA the objective.

3. BA produces requirements + acceptance criteria; runs its own checks; submits. UX/Dev/QA/Writer do an advisory read and file questions; BA resolves and baselines.

4. PM validates scope/priority/value, returns to BA if thin, else produces the backlog and dependency graph.

5. UX designs to the requirements; validates coverage + accessibility; baselines.

6. Solution Architect sets the system design and technical constraints (ADRs) from the requirements + UX; validates feasibility and NFRs; baselines — these become Development's inputs.

7. Development implements; passes build/lint/tests internally (hook gate) before submitting.

8. QA verifies against requirements + UX; returns defects to Development or passes.

9. Technical Writer documents the verified behaviour in house style; runs render + layout QA; submits.

10. Coordinator confirms every artifact is Approved/Baselined and closes the item.

> **What the pilot proves.** That contracts carry enough for each agent to work without back-channel instructions, that rejections route correctly, and that mechanical gates fire. Tune the charters from what you observe, then add roles.

11 Extending the system

New specialists slot in under the same ownership and handoff rules — no change to the model.

- **Security** — owns threat models and security acceptance criteria; advisory review of requirements, UX, and code; files findings back to owners.
- **Legal / Compliance** — owns regulatory constraints and clearances (a natural gate before anything external ships).
- **Operations** — owns runbooks, deployment, and observability requirements.

Each gets a charter (owns / inputs / authority / validation / corrections / model / tools), a place in the flow, and — where boundaries matter — a scoped tool set and a hook gate. The Coordinator and the contract never change; that's what lets the system scale without renegotiating how agents work together.

12 Maturity roadmap — treating the system like a product

The operating model is run the way any product should be: with an honest gap assessment and a prioritised roadmap. These are the known weaknesses and the planned hardening, in priority order.

- **Instrumentation & baselines.** STATE.md already tracks the right signals — first-pass yield, rework rate, rejection count, cycle time — but nothing yet baselines or trends them. Logging them per work

item creates a tuning loop for the iteration caps (are two cycles the right cap, or three?) and makes the system's ROI provable rather than anecdotal.

- **Universal mechanical enforcement.** Section 9's honest caveat — that the core rules are convention — gets closed by graduating them to mechanisms: write-scoping and pre-write hooks on every agent (not only where boundaries obviously matter), and schema validation on every handoff contract, so a malformed or authority-violating contract is rejected by a script before any agent exercises judgment on it.
- **Structured state.** Free-form markdown is fragile shared memory: one bad edit drifts the whole orchestration. Lifecycle state moves to structured data (YAML/JSON) updated through a small CLI that enforces legal state transitions; the human-readable STATE.md board is rendered from it, not hand-edited.
- **An agent regression harness.** A charter or steering edit is an untested deploy. A suite of golden work items — small, representative tasks with known-good outcomes — is replayed after any prompt change to catch behavioural drift (an agent that stops rejecting authority breaches, a reviewer that starts adding scope) before it reaches real work.
- **A post-ship outcome loop.** The pipeline verifies that work is built right; it does not yet verify it was worth building. An outcome-review step compares shipped results against the success metrics defined in the product brief and feeds the findings back into the roadmap — closing the loop from delivery back to direction.
- **Periodic model re-bidding.** Right-sizing decays: models improve and cheapen continuously, so an assignment that was optimal at build time won't stay optimal. Tracking cost-per-artifact per agent puts model assignments on a review cadence, with the regression harness (above) as the safety net for demoting an agent to a cheaper model.

> **Why publish the gaps?** Because the difference between a demo and an operating model is knowing exactly which guarantees are real, which are convention, and in what order the conventions get hardened. A system whose weaknesses are mapped and prioritised is a system being managed — not merely used.