

Agentic Development Workflow — Design Notes

Shawn Ohn • developer.ohn@gmail.com • Excerpt from a working engineering ADR, March–September 2026

Sanitized for sharing: company, vendor, and infrastructure identifiers removed. All figures are from recorded runs.

1. Context and goal

I am the sole engineer on a ground-up replatform of a live e-commerce marketplace (legacy Django/React to a TypeScript monorepo on NestJS, React, and PostgreSQL). To move faster without lowering the bar, I built a harness that lets a coding agent (Claude Code, run in the background) execute well-scoped tasks unattended — schema migrations, rename passes, fixture additions, per-page migrations — while humans keep every decision that requires judgment.

The governing document is an Architecture Decision Record that has gone through twelve revisions. Every revision was triggered by an observed failure, encoded as a numbered lock, and carried into the next dispatch so the same failure does not recur. This excerpt summarizes the parts of the design that answer three questions: *how is agent output verified*, *where does human approval sit*, and *how do we know the workflow is getting more reliable rather than less*.

2. Dispatch structure: what the agent is given

Each autonomous run receives a dispatch prompt assembled from a versioned, ten-section boilerplate. The structure is the product of the failure analysis described in §5; its sections are:

- **Pre-flight verification.** Three shell checks (`grep/wc/find`) with expected values, run before any edit. Mismatch means abort and write a failure report — never “proceed and hope.” Expectations are written as *intent plus tolerance* (“7 files, or ≤ 9 files with combined table count 8”) after a run correctly refused to abort on a benign count difference and documented it instead.
- **Write cadence.** An explicit four-step loop per unit of work: `edit` → `git add` → `git commit` → append a timestamped line to a progress file. Stated as a loop, not a preference; see §5 for why.
- **Done report.** A file written to disk — not a chat message — listing commit hashes, tests run (count), context utilization, and every acceptance claim. Written at completion *and* proactively at 50% context usage, because agents defer file writes in the cycle right before their context is compacted and the report is lost.
- **On-disk verification.** Every acceptance claim must quote raw command output. “Verified all 7 sections filled” is rejected; “`grep -c MARKER` → 0, output below” is accepted. If a verification is not possible within budget, the agent must say so rather than infer.
- **Handoff protocol, time limit, DDL patterns, CLI gotchas, network resumability, environment parity** — operational sections accumulated from specific incidents.

Five reusable prompt templates exist by task shape (ALTER migration, greenfield table set, mixed, net-new domain, frontend per-page). Project-level instructions live in a per-app `CLAUDE.md` that every run reads first.

3. Verification: the frozen evaluator

Every run ends by passing a fixed evaluator before its branch can be merged. “Frozen” means the agent cannot modify the evaluator, its thresholds, or the test inventory it is graded against; those are human-owned.

Gates

`lint` → `typecheck` → `build` → `db reset` (migrations apply cleanly from zero) → tests. A fail at any stage is a red run regardless of what the agent reports.

Anti-gaming checks

The suite being green is a *measure*, and an agent optimizing for green will find the cheap routes to it. Each of the following closes one:

- **Diff size cap** — changed lines must stay under 200% of the task’s estimate; a shrinking or exploding diff is flagged. Catches tests being deleted to make failures disappear.
- **Mock ratio** — mocked dependencies under 30% of test surface. Catches suites that pass because every real dependency was replaced by a stub returning what it was told to.
- **Skip count = 0** — no `test.skip`, no silently disabled cases.
- **Tests-run count > 0** — a test command can exit 0 having run nothing (the `-passWithNoTests` trap). The run is red unless the task carries an explicit `tests_required: false` flag, which only scaffolding tasks may set. This one was found in the wild, on the first production run, not anticipated.

- **Context < 90%** — treated as a scope-creep signal; a task that needs more than this is split rather than pushed through.

The acceptance layer itself was written *before* implementation: a 280-case test inventory, organized by domain and priority, that the agent is graded against but cannot edit.

4. Human approval boundary

What agents may do

Schema migrations from a written spec; repo-wide rename passes; golden-fixture additions; per-page migration of read-only routes once a human has established the page pattern; documentation generation from existing content; type-drift fixes.

What stays human

New schema design; architectural “X or Y” decisions; first-of-kind foundations (the first page of a new pattern is always built by hand, then the agent applies it to the rest); ADR revisions; and **anything on the live money path**. Payment-provider endpoints are on the agent’s network blacklist, not merely discouraged.

Review and merge

Agents work in isolated git worktrees on their own branches. Nothing reaches main without a human reading the diff, the Done report, and the progress file together — the three are deliberately redundant so that a gap in one is visible in the others (one run verified as “7-line progress file ↔ 8-commit chain ↔ Done report”). Merge is a human action.

5. Measuring reliability: a controlled comparison

The clearest evidence that the workflow works came from a failure. The same documentation-consolidation task was dispatched twice:

	Part 1	Part 2
Prompt	stated “write per file”; 90-minute limit stated	added the explicit 4-step loop; report threshold lowered to 50%
Outcome	5 of 12 units silently failed	7 of 7 units clean
Runtime	≈9 hours (ignored the stated limit)	7 minutes
Audit trail	none written	8 commits + progress file + Done report

The only difference between the two prompts was the set of clauses that became locks R8-1 through R8-5. That diff is clean data on *which instruction prevented which failure*, and it produced two durable rules: an agent will not self-enforce a time limit (external supervision is required), and a cadence stated as a preference is not followed, while one stated as a loop is.

Convergence

After round 10, three consecutive production runs (catalog, cart/delivery, promotion/content, notifications domains — 12 to 16 minutes each, 9 to 15 commits each) produced **zero new locks**. The feedback loop had stabilized: the harness was no longer discovering failure modes faster than it closed them. A later run did surface a new class (a network drop mid-task, from which the agent cannot self-recover), which became its own lock with two documented recovery paths — consistent with the design’s premise that the list of failure modes is finite and enumerable, not that it is complete.

6. What I would tell a team adopting this

- Write the acceptance layer first and keep it out of the agent’s reach. Verification the agent can edit is not verification.
- Treat the context window as an operational resource with alarm thresholds. Most silent failures happened near compaction.
- Make the agent cite raw output for every claim. The cost is a few lines per report; the benefit is that a human review takes minutes instead of a re-run.
- Decide what not to automate before deciding what to automate. The boundary is the part of the design that makes the rest safe to use.
- Expect the rule set to converge. If every run is still producing new rules after a dozen, the task scoping is wrong, not the agent.