My AI-Native Software Development Workflow

*How I use coding agents to plan, write, test, review, and ship software **without giving up engineering judgment***

AI can write the implementation. I still own the problem, the architecture, the review, and the result.

I have been a software engineer for over 6+ years (at Amazon for 5 years), and today I use terminal-based agents such as Claude Code and Codex to produce **all of my code**. This is not the same as blindly prompting until something appears to work. The workflow has explicit planning, testing, diff review, independent AI review, and human approval gates.

The goal is not to preserve typing code as a ritual. **The goal is to deliver correct software faster while keeping engineering judgment where it belongs: with the engineer.**

The Workflow at a Glance

1. Define the problem, constraints, acceptance criteria, and risks.
2. Choose direct execution for a small change or plan mode for a larger one.
3. Have the coding agent implement it in small, reviewable increments.
4. Require meaningful tests and run the project's full verification toolchain.
5. Review the diff yourself locally and again in a pull-request or code-review surface.
6. Run a separate AI reviewer agent with fresh context.
7. Send valid review comments back to the coding agent, then re-test and re-review.
8. Merge only when CI, human review, and the final engineering bar all pass.

This is a loop, not a straight line: plan → implement → test → inspect → review → revise → verify.

A failed check sends the change back to the relevant earlier step.

Manage Agent Sessions With Herdr

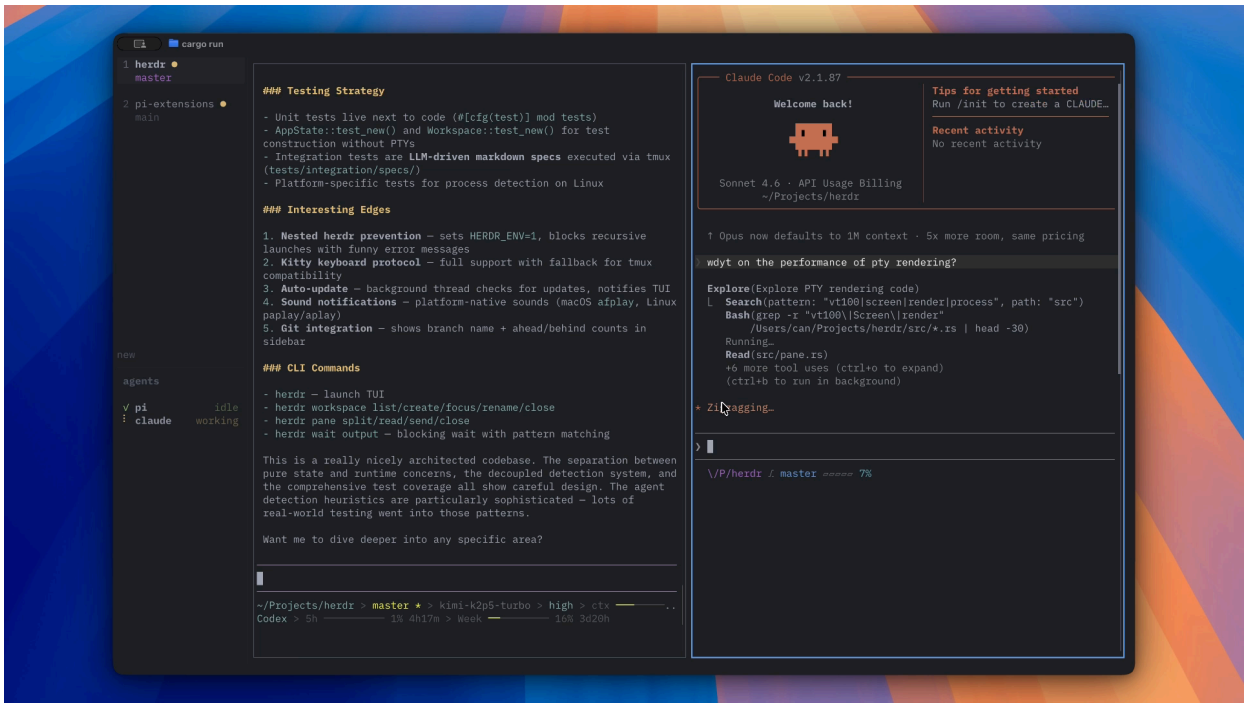
I use Herdr to manage multiple coding-agent sessions in a single terminal workspace. It lets me keep the implementation agent and the independent reviewer in separate sessions, while switching between their panes from one place.

Herdr runs the terminal processes in a background server. Closing or detaching the client does not have to end the work. I can reconnect later and pick up the same running sessions, as long as that server and its host are still running.

What Herdr adds to the workflow

- Organize agents, shells, tests, and development servers into workspaces, tabs, and split terminal panes.
- See agent activity in the sidebar and inspect the actual terminal output when a session needs attention.
- Detach the terminal client and reconnect without restarting the server-owned processes.
- Use the CLI or socket API to create panes, start agents, send prompts, read output, and wait for agent states. A script or lead agent can coordinate that loop.

Herdr provides the running environment and controls. The task breakdown, review criteria, and decisions about what to do next still come from me or a controller I configure. Herdr's working, blocked, idle, or done label tracks an agent's lifecycle state; that label does not establish that its code passed review or tests.



What stays alive when I step away

Detaching the client or losing an SSH connection leaves the server-owned processes running on their host. That is useful when I am away from the terminal. Work can still pause if an agent is waiting for a permission prompt, missing input, or a service that is unavailable.

For work to continue while my laptop actually sleeps or shuts down, Herdr and the agents need to run on a separate always-on machine, with the laptop attaching over SSH. A background server on the laptop cannot keep executing while the laptop itself is asleep. Being idle or turning off the display is different from system sleep.

A server restart is also different from detaching. Herdr can restore workspace layout and, when enabled, recent screen history; supported agent conversations may resume with the relevant integrations installed. The original processes have ended; this is recovery, rather than uninterrupted execution.

Herdr references: Agents · Agent automation · Persistence and remote access · Session restore · System sleep

1. Start With the Problem

Before I ask an agent to touch code, I make the task concrete. A model can implement the wrong thing extremely quickly, so problem definition is the first quality gate.

- State the user or business outcome, not only the code change.
- Identify the relevant repository area, existing patterns, constraints, and non-goals.
- Write observable acceptance criteria: what must be true when the change is complete?
- Call out risk: security boundaries, data migrations, compatibility, scale, or operational impact.
- **Ask the agent to inspect the current implementation before proposing a solution. This will help it build context**

Example: “Understand how the current payment system is working and then reduce duplicate payment attempts after an ambiguous timeout” is a stronger task than “refactor the payment service.” The first defines an outcome; the second can produce a clean diff that solves nothing.

2. Choose Plan Mode or Direct Execution

Use direct prompting for small, bounded changes

For a localized bug fix, small endpoint, validation rule, or clear refactor, I usually prompt the agent directly. The task should be narrow enough that I can understand the expected diff before it is generated.

- Name the behavior to change and the files or subsystems likely involved.
- State what must not change.
- Require tests with the implementation.
- Ask the agent to summarize the finished diff and any assumptions.

Use plan mode for longer or higher-risk asks

For multi-file features, unfamiliar systems, migrations, architectural changes, or work with meaningful failure modes, I ask for a plan before allowing edits. Claude Code has a plan mode, and Codex can be asked to investigate and produce an implementation plan before execution.

- Have the agent map relevant components, interfaces, data flow, and dependencies.
- Require a file-by-file change outline and a test strategy.

- Ask for assumptions, alternatives, rollout concerns, and open questions.
- Edit the plan. Remove unnecessary abstractions and challenge risky steps.
- Do not approve the plan until you can explain it yourself.

Rule of thumb: If a wrong implementation would create a large diff, touch data, change a public contract, or be hard to roll back, **plan first.**

3. Let the Agent Implement, but Keep the Loop Tight

Once the direction is clear, I let Claude Code or Codex edit files and run commands in the terminal. I do not wait until the end of a large task to look at the result.

- Work in small, coherent increments that could be reviewed independently.
- Ask the agent to follow **existing repository conventions before inventing new ones.**
- Keep the diff focused. Remove unrelated cleanup, generated filler, and speculative flexibility.
- Interrupt when the agent goes in the wrong direction; correct the instruction instead of hoping the next prompt fixes everything.
- Ask for a short change summary after each meaningful increment: **files changed, behavior added, tests written, and remaining risks.**

The interaction is iterative. I prompt, inspect, clarify, and reprompt until the implementation matches the intended design. I am not trying to write every line through natural language; I am steering the system toward a result I can defend.

4. Require Tests and Automated Checks

Tests are part of the implementation, not an optional cleanup step. The writing agent should add or update tests, but it should not decide by itself that those tests prove the change is correct.

- Test observable behavior and requirements, not only internal method calls.
- Cover the happy path plus relevant edge, failure, authorization, concurrency, and integration cases.
- Run the repository's formatter, linter, type checker, unit tests, integration tests, and build.
- Use static or security analysis where the risk justifies it.
- Manually exercise the critical path when automated tests cannot fully establish confidence.

Important: Green AI-generated tests are evidence, not proof. A model can write an implementation and tests that share the same mistaken assumption.

5. Review the Diff Yourself

I review the change in two places because they optimize for different kinds of thinking.

Use the diff tool that works best for you

The specific interface matters less than whether it helps you review the change carefully. Use the diff tool you are most comfortable with—VS Code or another text editor, a dedicated diff application, a pull-request interface, or terminal tooling such as git diff.

- Use an editor when inline diffs and surrounding code make the change easier to understand.
- Use terminal tooling when you want a fast, focused view of working-tree or staged changes.
- Use a PR or code-review surface when you need durable comments, CI status, and collaboration.

The tool is only the viewing surface. The review still requires you to understand the behavior, follow the data flow, and inspect surrounding code when a hunk is unclear.

Local terminal review

First, I inspect the working tree close to the agent session. I usually begin with the shape of the change, then move into individual hunks.

```
git status --short
git diff --stat
git diff
git diff --staged
git diff main...HEAD
```

- Check for surprising files, generated artifacts, lockfile changes, or accidental deletions.
- Review the diff by responsibility, not mechanically from top to bottom.
- Open surrounding code when a hunk cannot be understood in isolation.
- Use git add -p when you need to separate valid changes from noise.

Pull request or code-review review

Then I review the committed change in a durable review surface such as a GitHub pull request, GitLab merge request, Bitbucket pull request, Gerrit change, or an equivalent internal tool. These surfaces make file-by-file review, inline comments, CI state, and reviewer discussion easier to track.

At Amazon, I use Amazon's internal code-review tooling rather than GitHub pull requests for internal work. The mechanics are the same: inspect the diff, leave or resolve inline comments, verify automated checks, request human review, and iterate before approval.

- Read the PR description and confirm it describes the actual problem and solution.
- Review commits and the final combined diff; do not rely only on the agent's summary.
- Separate blocking correctness issues from optional style preferences.
- Re-review changed hunks after each revision, not only the newest comment thread.

6. Run an Independent AI Reviewer

I run a reviewer agent that is **separate from the agent that wrote the code**. This is deliberate: the writer has context and assumptions that can cause it to defend its own approach or miss the same flaw twice.

- Use a different review product, model, or at minimum a fresh session with no implementation conversation.
- Give the reviewer the problem statement, acceptance criteria, repository instructions, and complete diff.
- Ask it to prioritize correctness, security, reliability, architecture, and missing tests over cosmetic preferences.
- Require each finding to name the affected file or behavior, explain the impact, and propose a concrete fix.
- Treat every AI comment as a lead to validate, not an instruction to obey.

A separate agent improves independence; it does not create true human independence or guarantee correctness. It is another filter in the system.

7. Resolve Review Comments Through the Coding Agent

When the reviewer leaves comments, I triage them before changing code.

1. Validate the finding against the requirement, code, tests, and documentation.
2. **Reject false positives or low-value churn with a short reason.**
3. Group valid comments by root cause instead of fixing symptoms one at a time.
4. Send the accepted comments back to Claude Code or Codex with the relevant context.
5. Ask the writing agent to explain its proposed fixes before applying risky changes.
6. Run focused tests, then the full verification suite.
7. Inspect the new diff and re-run the independent reviewer when the revision is material.

Do not do this: “Fix every reviewer comment.” That instruction allows one model to blindly implement another model's mistakes. You remain the arbiter.

8. Final Human and Production Gate

The workflow ends only when I can own the change—not when the agent stops responding.

- CI is green and the important test results are understood.
- The final diff is focused and every meaningful file is accounted for.
- Compatibility, rollout, migration, observability, and rollback are handled where relevant.
- A human teammate has reviewed the change when the project or risk level requires it.
- Documentation, runbooks, and public contracts are updated when behavior changed.

- I can explain the main components, high-risk decisions, failure paths, and why the implementation solves the original problem.

Useful Additions When Scaling the Workflow

The core loop above is how I work. The practices below are recommendations for making parallel and unattended work easier to trust. Apply them where the project and its risk justify the extra setup.

Give parallel work clear boundaries

- Give each concurrent writing agent its own Git worktree and branch. Separate terminal panes do not isolate files. Agree on interfaces before splitting tightly coupled work, and have one owner integrate the changes.
- Worktrees isolate checkouts, not every shared resource. Give local servers distinct ports and use separate test data where needed. Re-run integration checks on the combined branch.
- Hand the reviewer a specific commit or diff, the acceptance criteria, and relevant test results. Keep its role review-only. After fixes, review the updated revision so approval applies to the code being merged.

Keep context and progress recoverable

- Keep concise repository instructions in AGENTS.md or CLAUDE.md: setup, test commands, conventions, constraints, and risky areas. Use the repository and the agreed plan as the source of truth when sessions restart or context is compacted.
- At each useful checkpoint, record the branch or commit, files changed, decisions, checks run and their results, blockers, and next step. Set a time or cost budget and a clear condition for stopping or asking for help.

Keep the release gate outside the agent

- Give agents only the filesystem, tools, and credentials the task needs. Keep secrets out of prompts and commits, and treat instructions found in issues, websites, and tool output as untrusted input. Require explicit approval for destructive or production-changing actions.
- Have CI verify the final integrated revision with deterministic checks. Preserve test output and explain skipped checks; an agent's success message is not the release record.
- After deployment, run a smoke test of the real user path and check relevant errors or metrics. For risky changes, define the rollback trigger and owner before shipping.

Further reading: Git worktrees · Claude Code best practices

Tool Options at Each Stage

You do not need this exact stack. The useful pattern is agent → tests → human diff review → independent reviewer → revision → final human gate. Some options below are open source; others are hosted or commercial.

Stage	Strong default	Alternatives
Agent sessions	Herdr	Separate terminal sessions; another terminal workspace manager
Plan and implement	Claude Code, Codex CLI	Googles gemini, Deepseek, Kiro CLI
Plan mode	Claude Code plan mode	OpenCode plan agent; A fresh read-only agent session
Local diff	Git diff	Delta (open source); Difftastic (open source)
Review surface	GitHub pull requests	GitLab merge requests; Bitbucket pull requests; Gerrit or internal tooling
Independent AI review	CodeRabbit, Greptile	Qodo PR-Agent (open source); GitHub Copilot code review; fresh Claude/Codex reviewer
CI and final gate	Your repository CI	GitLab CI/CD; Buildkite; Jenkins

Open-source labels refer to the tool implementation, not necessarily the models, hosting, or API access used with it. Always verify your employer's security and data-handling rules before giving an external agent access to private source code.

The Engineer’s Guide to Reviewing AI-Generated Code

See Engineer's Guide to Reviewing AI-Generated Code for a full guide on how to review Ai generated code like an engineer!

Reusable Prompt Templates

Planning a larger change

```
Inspect the repository before making changes. Create an implementation plan
for [goal].
```

```
Include: current architecture and relevant files, assumptions, proposed
design, file-by-file changes,
test strategy, failure modes, security/compatibility concerns, rollout and
rollback, and open questions.
Do not edit files yet. Prefer the simplest design that meets the
requirement.
```

Implementing a small feature

```
Implement [specific behavior]. Keep the change limited to [scope]. Preserve
[existing contract].
Follow repository conventions and avoid unrelated refactors or new
dependencies.
Add behavior-focused tests for [cases]. Run [commands].
When finished, summarize files changed, assumptions, verification results,
and remaining risks.
```

Requesting rigorous tests

```
Add tests that prove the observable requirements rather than mirror the
implementation.
Cover the happy path and the relevant boundaries, invalid inputs, failures,
authorization, concurrency,
and integration behavior. Explain what regression each test would catch.
Avoid mock-only assertions
unless the interaction itself is the contract.
```

Running the independent review

```
Act as an independent senior code reviewer. You did not write this change.
Review the complete diff against the problem statement and acceptance
criteria.
Prioritize correctness, edge cases, security, reliability, architecture,
performance, API/version accuracy,
missing behavior-focused tests, and production readiness. Ignore trivial
style unless it harms maintainability.
For each finding, provide severity, file/location, impact, evidence, and a
concrete recommendation.
Do not assume the implementation or its tests are correct.
```

Resolving validated comments

```
These review findings have been validated: [paste accepted findings].
For each one, explain the root cause and proposed fix before editing.
Implement the smallest coherent correction, update behavior-focused tests,
and run the full verification suite.
Do not change unrelated code. Summarize how each finding was resolved and
flag any trade-offs.
```

Final verification

```
Review the final branch against the original goal and acceptance criteria.
List any remaining correctness, security, reliability, compatibility,
testing, rollout, or operability risks.
Confirm the diff is focused and that every changed file is necessary.
Do not edit code. If you find nothing, say what you checked and what
remains unverified.
```

Handing off parallel or unattended work

```
Work on [task] in [worktree and branch]. The scope is [allowed files and
behavior].
Do not edit another agent's working directory or change production state.
Use [acceptance criteria] and run [verification commands]. Stop for
[approval conditions].
Before handing back, report the commit or diff, checks and their outputs,
unresolved risks,
and the next step. The reviewer should evaluate that revision in a fresh
session.
```

The Principle Behind the Workflow

Software engineering was **never only about typing code**. It is about understanding a problem, making trade-offs, building the right system, and owning what happens after it ships.

Agents are now one of the strongest implementation tools we have. Use them aggressively, but surround them with planning, tests, independent review, and human judgment**. The agent can produce the code. You are still responsible for the software.**

Final Note

Follow @daniallqureshi for more useful resources and tips!!

https://www.instagram.com/daniallqureshi/

https://www.tiktok.com/@daniallqureshi

https://www.youtube.com/@daniallqureshi

https://x.com/daniallqureshi

https://linkedin.com/in/danial-qureshi