Enforcing Engineering Discipline with AI

The Lifecycle Hasn't Changed. Now It's Enforceable.

Alex Djalali
Research Faculty, Georgia Tech
Former Software Architect & Principal Engineer (Salesforce & Tableau)



The Software Engineering Lifecycle

Every rigorous engineering team follows the same phases:

Architecture → Design → Plan → Implement → Test → Verify → Ship

> The question isn't whether this lifecycle matters. It's whether we actually follow it — every time, under pressure, at speed.

The Risk: Vibe Coding

Skipped Lifecycle

- "Build me an app" → ship it
- Architecture? AI decided.
- Design? Skipped.
- Tests? Maybe later.
- Review? "It works."
- **Tech debt at machine speed**

Enforced Lifecycle

- Architecture → human decision
- Design → human decomposition
- Plan → human approval
- Implement → TDD, AI executes
- Verify → gates, non-negotiable
- **Leverage at machine speed**

AI makes it trivially easy to skip every phase. That's the risk — not that AI writes bad code, but that it lets you bypass the process that catches bad decisions.

The Thesis

> AI doesn't replace the engineering lifecycle. It **enforces** it.
>
> Your rigor, encoded and automated.

- You already know what good engineering looks like
- The problem was never knowledge — it was **consistency**
- AI is the mechanism that makes your standards **non-negotiable**

> *"The source of leverage isn't the AI. It's the constraints you put around it."*

Decide: Architecture, Design & Planning

The lifecycle starts with decisions, not code.

- **Architecture**: ADRs capture the *why*. AI explores and proposes — **humans decide and own.**
- **Design**: Epics decomposed into stories with clear scope. AI identifies sub-tasks — **humans set boundaries.**
- **Planning**: AI reads the codebase, proposes concrete tasks. **No code until a human approves the plan.**

> **Principle:** Judgment stays with the engineer. AI accelerates exploration, not decision-making.

The traceability chain starts here — every PR links back to the ADR and plan that motivated it.

Build: Implementation & TDD

TDD — failing test first. Always.

1. **Red** — Write a failing test that encodes the requirement
2. **Green** — Write the minimum code to pass
3. **Refactor** — Clean up while tests stay green

AI-generated code is a **statistical approximation** of what you asked for.
Tests are **encoded invariants** — they don't approximate, they assert.

Without TDD

- AI writes plausible code
- Bugs hide in edge cases
- Refactoring is risky

With TDD

- Test encodes the requirement
- Edge cases tested explicitly
- Refactoring is safe

Ship: Verification & Delivery

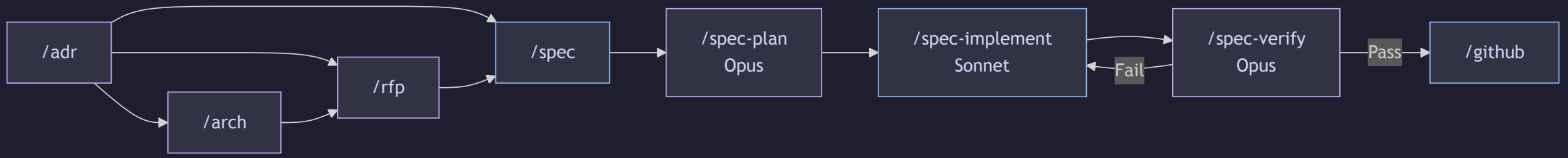
Four gates before every commit. No exceptions.

Gate	What it catches	Auto-fixable?
Format	Style inconsistencies	Yes
Lint	Code smells, anti-patterns	Mostly
Type Check	Type errors, interface mismatches	No — AI fixes
Tests	Behavioral regressions	No — AI fixes

Same bar regardless of who wrote the code. Human or AI — the gates don't care.

Then: conventional commits, PRs that link to the plan, traceability from **Decision → Epic → Story → Plan → PR**.

The Full Lifecycle



Every phase connected. The **verify → implement** feedback loop is the key — failures don't end the process, they feed back into it.

Enter anywhere: small feature → plan directly. Structural change → start at architecture.

Encoding Your Standards

The mechanism is simple: **write your standards in a file AI can read.**

```
~/.claude/CLAUDE.md          → Standards for Claude Code
~/.cursor/rules              → Standards for Cursor
project/.claude/settings     → Project-specific overrides
```

One file. Readable by any AI tool. Contains:

- Language conventions, naming patterns
- Architecture rules, dependency constraints
- Quality thresholds, anti-patterns to avoid

> "*Structure beats intelligence.*"

Reference Code: Solving the Cold Start

Rules tell AI what to do. Reference code shows it *how*.

The 0→1 problem is real: on a greenfield project, AI has no patterns to follow. The fix:

- **Pull in gold-standard repos** — your own past projects that represent how you build
- **Curate third-party examples** — open-source code with patterns worth following
- **Seed the codebase** — a small, well-structured module gives AI a template to extrapolate from

> **Standards + reference code = a codebase that teaches.** AI doesn't just follow rules — it pattern-matches against examples. Give it good ones.

One clean module is worth a thousand lines of documentation.

AI Amplifies What's Already There

Good Discipline + AI

- Explicit architecture → consistent codebase
- TDD → reliable implementations
- Quality gates → no regression
- Clear standards → reproducible output
- **Leverage at scale**

Bad Discipline + AI

- No architecture → inconsistent mess *faster*
- No tests → confident bugs
- No gates → tech debt at machine speed
- Vague standards → random output
- **Liability at scale**

> *"You play like you practice."*

AI doesn't fix a broken process. It accelerates whatever process you already have.

Actionable Takeaways

What you can do Monday morning:

1. **Write your standards down.** If they only live in people's heads, AI can't follow them.
2. **Seed with reference code.** One gold-standard module teaches AI more than a page of rules.
3. **Pick one quality gate** and make it non-negotiable. Start with formatting.
4. **Enforce TDD.** AI-generated code needs tests *more*, not less.
5. **Separate judgment from execution.** Humans approve plans. AI implements them.

> One standards file + one reference module. That's the minimum. What took an afternoon is now **20 minutes** — but only if the discipline is there first.

The Lifecycle Hasn't Changed.

AI Just Makes It Enforceable.

Dotfiles: github.com/alexdjalali/dotfiles

`.claude/CLAUDE.md` · `standards/` · `templates/` · `commands/`

Questions?