



Developer Track

Crafting AI-Generated Code into reliable software - Session I

Bryon Jacob

Summer AI Intensive
August 2026

vibe coding

The original idea: you give the AI inputs, and it writes the code.

It's real, and it works — people build things this way every day

But it has limits

Today is about how building *real* software differs from vibe coding as it's usually understood.

The ROI gap

Most teams report little to no ROI from AI coding tools

About 5% report large gains

Both are real. The difference isn't mainly the tool — or even the team's expertise.

It's what the tool is pointed at. The quality of the codebase is probably the biggest factor.

First — tell us about yourself

Go to: <https://aaia-2026-bkj.val.run/>

Answer a couple of quick questions. This sorts everyone into three groups.

An experiment

Everyone starts from a codebase, with the same task.

Your task

Add a feature to the app — and keep it working.

The feature working is only half of it — **nothing else should break.**

Getting started

Fork your group's repo, then get it running:

```
# fork + clone (group A shown – use -b or -c for your group)
gh repo fork bryonjacob/aaia-summer-2026-a --clone
cd aaia-summer-2026-a

make sync                # install dependencies
DEBUG=True make migrate  # set up the database
make run-debug           # run it → open the URL it prints

claude                   # open Claude Code in the repo, and go
```

How will you know you succeeded?

Does the new feature **work**?

Did anything **else break**?

Could you tell either way — how?

Your confidence it's correct: ☐ low · ☐ medium · ☐ high



Go.

Add the feature · keep it working · score yourself

You did **not** all start from the same codebase

Same app — a real open-source project: <https://github.com/c4ffein/realworld-django-htmx>

We cloned it and handed out three different versions:

Group A — the repo *exactly as we cloned it*

Group B — a cleaned-up version: same features, but reorganized code and a lot more testing

Group C — same as B, plus an opinionated **AI-readiness framework** installed to manage how the work gets done

The point of the experiment

How much variance do we see in the ability to **add and manage a complex feature** — across the same app, the same task, the same AI, with only the codebase changing?

Same AI. Same prompt. Same you. **Different results.**

| AI punishes **messiness** and rewards **discipline**.

Developer Track

Crafting AI-Generated Code into reliable software - Session II

Bryon Jacob

Summer AI Intensive
August 2026

The economics flipped

Old: humans write all the code → cleanup barely pays back

New: AI writes most of the code — if the codebase lets it

The cleanup work? Exactly what AI is best at.

Discipline is now your highest-ROI investment.

How it works — spec-first, in small loops

	Phase	Who
Plan	You + AI — define the <i>what</i>	
Implement	AI — autonomous, tight loops	
Review	A <i>second</i> AI — adversarial	
Oversight	You — architecture & edge cases, not syntax	

You design the mold. AI fills it. The gates harden it — or reject it.

Round two

Everyone starts from the C codebase now.

What we'll do from here

Everyone forks the **C** repo: `bryonjacob/aaia-summer-2026-c`

Add another feature or two

Look at the [AI-readiness tooling](#) that makes it go

Why the C codebase works

A few principles behind the AI-readiness tooling.

Persistent project context

Every session, the AI starts blank — it doesn't remember last week.

But every project has standing facts: the stack, where things live, the conventions, what's load-bearing.

You put those in a **persistent context file** the AI loads automatically — so you don't re-explain the project every conversation.

Hierarchical and modular context

One giant context file is wasteful — the AI loads everything, even for one tiny corner.

Better: [a context file at every level](#).

Root — overview and conventions that apply everywhere

Each module — what it is, its boundaries, its key decisions

Deep areas — only the details that matter there

The AI loads what's near the file it's working on. Locality of context.

Tests are what make delegation safe

The kind that matter for AI: **fast, deterministic, automated** — thousands in seconds.

AI makes a change → run the tests → green or red → loop. A 20-minute suite kills the loop.

Coverage is how much of your code the tests exercise. Low coverage = changes can break things silently.

High coverage (90%+) is the safety net that lets you hand work to the AI and trust the result.

Enforced, not aspirational

"We aim for 96% coverage" is a wish.

"The merge is blocked if coverage drops below 96%" is a system.

Wishes drift. Systems hold. A quality bar isn't real until it's **automatic and unskippable** — same for linting, formatting, types.

Your codebase is the asset you own.

AI tools are rented leverage.

Make the asset valuable.