

IN THEIR OWN WORDS

How not to be an AI slop programmer.

We asked AAI Labs engineers one question in our team channel: how do you avoid being an AI slop programmer? Share one thing.

Here are the answers, lightly edited for clarity and grouped by theme. Names are left out on purpose. The one-page cheat sheet distils them into 12 habits; this is the long version.

41

REPLIES

37

ANSWERS
HERE

7

THEMES

0

MAGIC
PROMPTS

01 Understand the problem first

The most common theme. You can only judge the output if you know what the right answer looks like.

Know the domain well enough to push back.

Truly understand what you're supposed to do: the context, and what the end result should be. Without that you can't decide whether the generated code is good or aligned with what's needed.

Challenge the AI, question it, push back. That's how you get rid of the nonsense it produces. But it only works if you know the domain yourself. The code must be understood and verified by the developer.

ENGINEER

Focus on planning and validation. Think carefully about the domain, what's actually needed, and the edge cases, then test the generated code.

For bigger tasks I discuss the problem with AI and build a step-by-step plan before any code. It surfaces things I missed, gives the AI a clearer plan, and lets me validate each part separately.

ENGINEER

Understand the whole project and what you're aiming for, then use AI as an assistant. Understand what it's doing and guide it along the way.

I think of AI as a self-driving car. It can lose its way when there isn't enough data to guide it, so you need to know where you're going, and other ways to get there.

ENGINEER

When planning a new feature, I make sure I know what it should do. If I don't, I need more research and brainstorming, not more code.

In a new session, I have the agent review the project first (a shallow pass on large repos, then the relevant slice), and quiz it to make sure it isn't hallucinating. During implementation I start simple and minimal, then expand, so I never review hundreds of lines at once.

ENGINEER

Give it a detailed markdown file: your tech stack, your requirements, and what the finished product should do.

And keep it simple. AI tends to overcomplicate code, which makes bugs and errors harder to fix.

ENGINEER

02 Plan before you build

Several engineers spend most of their time here. The plan is where the expensive mistakes are cheapest to fix.

80% of my time goes into planning.

I spend 80% of my time in planning. Always use the best model for this, and refine the plan until you understand it.

To iterate faster, give the model a good verification loop: check web changes with Playwright, run QA, have one coding agent review the other's work, use sub-agents for review. After a good plan, review is easy.

ENGINEER

Write the plan yourself, then let AI review it.

Make the implementation plan yourself first. Then use AI to review it, improve it, and help with the implementation.

ENGINEER

I refine the plan until we're on the same page: edge cases handled, assumptions stated. After implementation I manually check that the feature works as expected.

ENGINEER

Use AI to validate or improve your proposed solution before implementation. Implement in smaller phases, and review and understand the code at each phase.

ENGINEER

I explain how I see the problem and my rough idea for solving it, then ask the agent to verify my thinking and suggest better approaches. I ask it to explain solutions as simply as possible, often with visual artifacts.

Then it implements, tests, and runs an adversarial review, including testing the feature. Finally I test it manually and read through the code once.

ENGINEER

Review the plan before implementation starts. Implement in stages, and review the changes after each milestone before moving on.

ENGINEER

Spec → Plan → TDD → Implement → Review → Push

ENGINEER

03 Make it ask, not guess

Most slop is the model filling gaps with confident assumptions. These habits close the gaps before the code is written.

Most slop is AI confidently filling gaps with assumptions.

Never let it guess. Make it read the actual source first (ticket, thread, code, live data) and push back when it assumes something it didn't check. Most slop I see is AI confidently filling gaps with assumptions.

Know what the expected output looks like before it starts, so you can check against it: is it like this? Why is it like that? And make it ask for your input instead of deciding by itself. Important decisions should always be yours.

ENGINEER

During planning, ask the agent to identify potential issues and unclear requirements. Address them before implementation starts. Otherwise they come back as review findings.

ENGINEER

- State the non-negotiable rules the agent must follow.
- Describe the problem in detail and tell it to ask questions or do web research.
- Question the plan and have it visualize it. For larger tasks, run a sub-agent for an unbiased audit of the plan.
- Verify the plan against reality. When integrating with an agent runtime, have it inspect the actual image for return formats and required parameters.
- Split the plan for TDD, read what it wrote, have it verify the result, then test it yourself.

ENGINEER

Have a general idea of the implementation, and ask the AI to ask questions rather than assume. Sometimes it thinks it knows what you meant and builds something entirely different.

ENGINEER

Resolve the task's open questions without letting the agent interpret them, for example with a /grill-me command.

When the task is done, review the code against the task in a new session. After a few rounds, review the final implementation yourself and look for the usual issues: useless or wrong tests, scattered imports, very long functions, rewritten code that already existed.

ENGINEER

- Tell it to ask you a lot of questions
- Ask it questions
- Use TDD
- Review in another session
- Understand what it's doing
- Have it verify everything live, by running and using it

ENGINEER

Size iterations to how much you can verify.

Size iterations to how much you can verify. Tools and skills make long agent runs safer, but with less certainty, shorter iterations and more frequent feedback make sense.

For completely unclear tasks, I start by finding the distinctions that change the expected result.

ENGINEER

04 Ground it in what exists

Reuse before writing. Evidence before opinion.

Make the AI explore the existing code before writing new code. Reuse what's already there and follow the project's patterns.

ENGINEER

Revert the fix and confirm the test fails.

Read the diff live. My instruction files say: targeted changes only, no unrequested changes, incremental updates over rewrites. When I see a big overwrite, poor naming or scope creep, I stop it immediately instead of cleaning up later.

Reuse before writing. Before coding, the AI must search the repo, read two or three similar features and copy their patterns, then tell me what it reused and what is new.

Tests must prove something. Test database instead of mocks where possible. Before a fix counts, revert it and confirm the test fails.

Turn corrections into rules. A repeated mistake goes into the repo's instruction file with the reason, so it doesn't come back next session.

ENGINEER

05 Small steps, minimal diffs

Keep every change small enough to actually review.

Break tasks into small, independently reviewable changes and have AI implement them one at a time. Reviewing and verifying behaviour gets much easier.

ENGINEER

Plan first, then build in stages scoped per feature rather than one large commit. Don't let the AI auto-commit; review the changes yourself before committing. I append "maintain KISS, DRY, SOLID and YAGNI" to push it toward the simplest path that works.

ENGINEER

If you can't defend it without the AI, it's not ready.

- Ask for proof, not "it works".
- Make it cut. Assume the first draft is a complex version of what's needed.
- Get a second model to review.
- Review in slices.

If you couldn't explain and defend it in the PR review without the AI, it's not ready.

ENGINEER

For anything new, challenge the AI to find evidence and sources for how it should be written. It leads to better decisions. Two public skills I use:

- **source-driven-development** (addyosmani/agent-skills): a light skill that makes the agent check sources, SDKs and docs.
- **Deep-Research-skills** (Weizhena): a heavy research framework for broad investigations. Watch the token use.

ENGINEER

I mostly work in notebooks, so I split the task into small steps and treat each AI-written cell as its own review unit.

ML RESEARCHER

Don't let AI change tests just to make them pass.

One small feature per call. Define tests before implementation. Don't let AI change tests just to make them pass. Ask for sources when researching.

ENGINEER

Enforce minimal code changes through prompts and skills, and challenge the amount of change proposed. Understand the underlying structure and how it will hold up to future changes, rather than every line.

Use multiple reviewers, with different skills and different models. Each is good at something, none at everything.

ENGINEER

06 Verify and own it

The final check is yours. The agent's summary is not evidence.

Review what it did, not what it says it did.

Don't review what it says it did. Review what it did. The end-of-task summary is the least reliable part; look at the diff, the logs and the data it touched.

ENGINEER

Question the AI. Interrogate the implementation and understand the assumptions it made.

ENGINEER

I made a slash command, /asking, that answers "how does XYZ work?" in three sentences or fewer. I ask about one small thing at a time until I fully understand it. It catches most slop before it ships.

ENGINEER

Take full responsibility, and be ready to explain it.

Take full responsibility for what you build, and be ready to explain it in detail. That alone makes you careful: you'll review the code, question the implementation and make sure the work is good.

ENGINEER

07 Judgment and taste

Two answers that zoom out.

Writing code can be delegated. Judgment can't.

I'd define slop in programming as code that's generated and accepted without enough judgment behind it. The common theme in this thread is more thought in planning and validation.

The programmer's role is shifting from producing code toward architect, decision maker, supervisor. That isn't a bad thing. It means providing enough context to guide the implementation and defining what evidence would justify accepting a change. Writing code can be delegated. Judgment can't.

ENGINEER

-
- Understand your pipeline: why method X and not Y. Ask the agent, and it often admits Y would have been better.
 - Spend a good part of your time in plan mode.
 - Keep a review skill with your project's rules. Once an agent quietly switched the vision model I'd specified; a verifier skill would have caught it.
 - Have the agent test the method on all available data, compare approaches quantitatively, and report the metrics.
 - Draw a diagram of your method to understand all its parts.

ENGINEER

Don't push a pull request without reviewing it yourself first. To make that concrete, I use a PR-ready checklist as a prompt (see the appendix).

ENGINEER

Add your own ideas, judgment and taste. Most AI-generated work has one thing in common: it feels generic.

Ask yourself: have I added something that someone who has never written a line of code couldn't have produced by prompting an AI?

ENGINEER

I don't code, but I end prompts with: "Do not rush. Work as much as you like. I'm not in a hurry." My subjective feeling is that results get better.

Another colleague's contribution, offered with a smile: just say "make no mistakes".

FROM OUTSIDE ENGINEERING

PR-ready checklist

A prompt one of us runs before opening a pull request. Copy it into your own agent setup.

DOCS

- Link the originating ticket or task.
- Summarize what changed and what it delivers.
- Outline code changes by module or file.
- Update docs and the changelog.
- Attach a brief ADR for significant features or architectural changes.

CODE

- Minimal diff: change only the files the task requires.
- Justify each new dependency: maintained, compatible license.
- 1–3 end-to-end behavioural tests in given/when/then style.
- Tests cover the main acceptance criteria and relevant failure and permission paths, at the lowest useful level plus the system boundary. Each must fail without the change.
- Avoid mocks. No trivial tests that only exercise mocks.

EVIDENCE

- **Acceptance table.** Restate the task as 3–8 concrete criteria and link proof for each: test name, screenshot or log line.
- **Live test** through the user-facing interface: screenshots or a recording for UI, a terminal transcript for CLI, request/response examples for APIs, plan/apply output for infra. State commit, environment, flow and outcome.
- **Exploratory testing** of the changed flow, one adjacent flow, and failure, empty, loading and permission states. List what was left untested.

DEPLOYMENT

- Confirm compatibility with existing data, config and integrations.
- Document migrations, deploy order and rollback steps.