A FIELD GUIDE

Design Before Generate

Using AI to write software without outsourcing the thinking that forms an engineer.

For interns, new grads, and junior engineers —
and the mentors and managers helping them grow.

A NOTE BEFORE YOU START

I wrote this after several conversations with early-career engineers who are entering the field at a very different time than my generation did. Some were interns. Some were new grads. Some were people close to me.

They are smart, capable, and already using AI coding tools as part of their daily workflow. That is not a bad thing. AI will be part of how modern engineers work.

But I care deeply that young engineers do not skip the part of software engineering that builds judgment. This guide is for them — and for the mentors and managers trying to help them grow.

The goal is not to use less AI. The goal is to use AI without outsourcing the thinking that forms an engineer.

Design before generate.

WHAT THIS IS

A loop you can run on every task

This is not a warning about AI. It is a practical habit — a short design loop you run before you prompt, so the solution takes shape in your head first and the AI becomes a reviewer of your thinking instead of the source of it.

You can read it in one sitting and use it the same afternoon. The heart of it is a single page: the loop on the next spread. Everything else is there to make that loop stick.

THE DISTINCTION THAT MATTERS

Reviewing code and designing a solution are different muscles

Here is a pattern worth naming. An intern told me her manager encouraged her to lean on the AI assistant and not worry too much about writing the code herself. A new grad described her actual workflow: the tool generates the solution, she reads it closely to understand it, then moves on.

Both are shipping. That is what makes it easy to miss. The ticket moves, the code compiles, the pull request opens. From the outside, everything looks like learning.

But **reviewing generated code builds recognition. Designing a solution builds judgment.** You need both, and only one of them comes for free when the AI goes first.

Recognition — understanding what exists

- “This function makes sense.”
- “I can follow how this API is called.”
- “I see what the generated code does.”

Judgment — deciding what should exist

- “This belongs in the service layer, not the controller.”
- “This interface is leaking implementation detail.”
- “This model will make the next requirement harder.”
- “This fails when the dependency is down.”

Judgment is the real apprenticeship of software engineering: decomposing the problem, deciding which layer owns which responsibility, defining the contract, reasoning through the sequence, seeing where the design fails, and knowing what tradeoffs you are making. AI can accelerate recognition. Judgment still has to be practiced.

THE HIDDEN COST

What gets skipped when AI goes first

When you start from AI-generated code, you can skip the blank-page loop — the one where you sit with the problem before a solution exists. That loop is where the core skills form: what am I solving, what are the components, where should the logic live, what are the contracts, what can fail, how will I test it, what am I trading off.

If the AI answers those first, you will still understand the final code — but you will not have built the same internal model. Do that for a few years and the gap becomes visible:

Engineers who can approve generated code, but cannot reshape it, challenge it, or own it when it breaks at 2 a.m.

That is not an AI problem. It is an apprenticeship problem — and apprenticeship is what this loop is built to protect.

THE FIX IS NOT LESS AI — IT IS SEQUENCE

The Design Before Generate loop

Before you prompt, attempt the design yourself. It does not need to be perfect. It does not even need to be right. The point is to force your own brain through the problem first — then bring the AI in as a reviewer, not the architect.

Steps 1–7 are yours, on paper, before any AI. Step 8 — the amber seam — is the reorder the whole habit turns on: only now does AI enter, and it enters to critique a design you already have.

THE LOOP, STEP BY STEP

Ten steps, one habit

PHASE 1 · DESIGN — THIS IS YOURS (NO AI YET)

1

Understand the problem

Write it in plain language: what behavior changes, who uses it, what “correct” means, what is out of scope. If you cannot explain it simply, you are not ready to prompt.

2

Sketch the end-to-end flow

Trace how the request moves through the system — from the user action to the response and back. Even if you only touch one layer, understand the ones upstream and downstream of it.

3

Place the responsibility

Which layer owns the business rule? Validation? Persistence? Orchestration? Many weak designs are not wrong because the code fails — they are wrong because a responsibility lives in the wrong place.

4

Define the contracts

What method, API, or event is needed; the inputs, the outputs, the errors, and what the caller should *not* need to know. Good engineering is often just boundary design.

5

Map the sequence

What calls what, in what order. What is synchronous, what is async, what state changes, where the flow can stop. A few boxes and arrows on paper are enough — the point is a mental execution model.

6

Name the failure cases

Invalid input. Unauthorized caller. A dependency that is down. A write that succeeds while the downstream call fails. The same request sent twice. Failure-mode thinking is what separates producing code from owning a system.

7

Decide what proves it

List the tests that would show the behavior is correct: happy path, invalid input, a permission case, the edge case that matters, the regression worth protecting. Naming the tests forces you to clarify the behavior.

> **WATCH THE SEAM**
>
> Everything above is yours — done before the AI sees anything. The next step is the reorder the whole habit turns on. This is the one place risk re-enters, so it is the one place to stay deliberate.

PHASE 2 · GENERATE — AI ENTERS AS A REVIEWER

8

Ask AI to critique the design

Now — and only now — bring in the AI. Not “write me a solution,” but “here is my design, what am I missing?” Ask it to challenge your layers, your interface, your edge cases, your coupling, and whether a simpler approach exists. You still own the thinking; the AI is a reviewer of it.

9

Implement with AI

Now let it accelerate — boilerplate, syntax, test scaffolding, refactors. The difference is that the code is anchored to a design you already reasoned through, instead of inventing one from a vague request.

10

Review, test & explain

Before you submit, be able to say: what changed, why it belongs in these layers, what pattern it follows, what the AI suggested, what you accepted or rejected, what proves it works, and what could still break. If you cannot explain it, you do not own it yet.

KEEP THIS NEARBY

The pre-flight checklist

Answer these before you ask AI to generate anything. It is a two-minute pass, not a document.

Problem

- [ ] What problem am I solving, and what does correct behavior look like?
- [ ] What is explicitly out of scope?

Flow & layers

- [ ] What is the end-to-end flow — what calls what?
- [ ] Which layer should own this responsibility? Which should stay untouched?

Contracts & data

- [ ] What are the inputs, outputs, and possible errors?
- [ ] What data is read, what is changed, what model or schema is affected?

Failure & tests

- [ ] What happens on invalid input or an unavailable dependency?
- [ ] What tests prove the happy path, the edge cases, and the regression?

AI critique

- [ ] What did the AI point out — and what did I accept, reject, and why?

A BETTER WAY TO PROMPT

Hand the AI a design, not a blank request

The weak prompt is “build this feature.” The strong prompt makes you think first — and it produces sharper feedback, because the AI is responding to a specific design instead of inventing one.

DESIGN-CRITIQUE PROMPT

I need to implement [feature / change]. My current understanding of the flow is [describe flow]. I think the responsibility belongs in [layer] because [reason]. The interface takes [input] and returns [output]. The main failure cases are [list], and the tests I plan to add are [list].

Critique this design. What am I missing? Where are the boundaries weak? Is there a simpler or more maintainable approach?

Notice what the prompt forces: you cannot fill in the blanks without having run the loop. That is the point.

FOR MENTORS & MANAGERS

Ask for the thinking, not just the code

“Use AI to move faster” is an incomplete instruction. The fuller version is: **use AI to move faster, and bring me your design thinking.** For interns and new grads, ask for lightweight design artifacts before implementation — not heavy documents, just enough to show a mind was at work:

- a rough flow sketch and the layers touched
- a proposed interface and its failure cases
- a short test plan
- a few sentences of reasoning in the pull request

A better pull-request standard

In the AI era, a strong PR shows more than changed files. It answers: what problem this solves, what design was chosen, which layers and contracts changed, what alternatives were considered, what the AI helped with, what the engineer verified by hand, and what the known risks are. That makes the engineer’s *judgment* reviewable — not just the generated code. The goal is not bureaucracy. It is apprenticeship.

BEYOND THE STACK

Functional understanding still matters

Being full-stack is valuable. But the future-relevant engineer also needs functional literacy — understanding the workflow, user need, business rule, or operational impact behind the code. Without it, AI will happily produce a solution that is technically plausible and functionally wrong.

For every task, it is worth asking: who uses this, what workflow does it support, what rule am I encoding, what exception matters, and what happens if this data is wrong. The strongest engineers will not simply be full-stack. They will be technically strong, system-aware, and functionally literate.

THE LONG VIEW

Why this is worth the discipline

As AI keeps getting better at generating code, the scarce skill moves upstream — from producing code to judging systems. That makes human engineering judgment more valuable, not less. The next generation of strong engineers will not be the ones who avoid AI. They will be the ones who can design first, use AI second, and still own the system when it breaks.

If you are early in this, I hope this saves you a few years of learning the hard way. Sketch the design before you prompt — rough, imperfect, probably wrong. It just needs to be yours. That first attempt is where the engineer in you gets built.

Design before generate.