Write It First, Then Let AI Drive

April 2026

9 min read • 2,115 words

There's a thing that happens when you start using AI coding tools seriously. You assume the best workflow is obvious: let AI generate the first draft, then you clean it up and maintain it by hand.

I've been finding the opposite to be true.

The Conventional Wisdom Is Backwards

The story most people tell about AI coding tools goes like this: let AI generate the boilerplate, the scaffolding, the initial version. Then you, the human, take over for the real work — the architectural decisions, the maintenance, the subtle taste judgments that make a library feel right.

It sounds reasonable. AI is fast at generating code. Humans are good at judgment. Let each do what they're best at.

I've been finding the opposite to be true. Write the initial version by hand. Pour your taste, your instincts, your opinionated architectural decisions into that first draft. Then let AI take the wheel for cleanup, feature additions, bug fixes, documentation, and ongoing maintenance.

> This might seem obvious in five years, or completely wrong. I'm reporting what I'm actually finding in practice, not what I think the theory should be.

This isn't a minor workflow preference. It's a fundamentally different model of what human-AI collaboration looks like in software, and I think it reveals something important about how intelligence — artificial or otherwise — actually works.

The Codebase Is the Style Guide

When I wrote Requests in 2011, I spent an unreasonable amount of time on the API surface. Not on the HTTP implementation — that was relatively straightforward. On how it felt to use. The names of things. The shape of the response object. Whether you'd write `response.json()` or `response.to_json()` or `response.as_json`. Those decisions weren't arbitrary. They encoded a specific philosophy about how humans think about HTTP.

That philosophy lives in the code itself. Not in a `CONTRIBUTING.md`. Not in a design document. In the actual patterns of how things are named, how errors are handled, how the public API relates to the internal implementation. The code is the style guide, written in the most unambiguous language possible: working software.

> There's a concept in philosophy called tacit knowledge — things you know but can't fully articulate. A hand-written codebase is tacit knowledge made legible. You couldn't write a document that captures every micro-decision, but the code captures all of them by definition.

When Claude Code reads a codebase I've written by hand, it's reading that embedded philosophy. It picks up on naming conventions, error handling patterns, how I structure modules, where I put comments, what I choose to make public versus private. It's not just reading syntax. It's reading intent.

And then it can extend that intent faithfully. When it added GZip middleware to Responder, it didn't invent a new pattern. It followed the pattern I'd established with the existing middleware stack. When it adds a new scale system to PyTheory, it follows the architecture I laid down for how musical systems are structured. The AI isn't making taste decisions. It's amplifying taste decisions I already made.

This is the key insight: **a hand-written codebase is a style guide written in code itself.** AI can read it and extend it. But if AI writes V1, there's no style guide to read. There's no human signature. There's no opinionated foundation. There's just... generated code, competent but characterless.

What V1 Encodes

Let me be specific about what gets embedded when you write the first version by hand, because it's more than just "code style."

Architectural instincts. Say you're building a music theory library. AI might start with `Scale` or `Chord` as the primary abstraction, because those are the higher-level concepts that appear most in documentation. But if you play guitar, you know that individual notes are what your fingers touch, and the library should build up from physical reality. That architectural choice shapes every feature that comes after it.

Taste in naming. `resp.html` instead of `resp.set_content_type_and_body("text/html", content)`. `Note("C4")` instead of `NoteFactory.create(note="C", octave=4)`. These naming decisions aren't cosmetic. They encode a philosophy about how humans should interact with code — a for humans philosophy. AI trained on the entire internet has seen every naming convention in existence. It doesn't have a preference. But once it reads a codebase with a strong naming voice, it can match that voice precisely.

> The "for humans" philosophy isn't just about readable APIs. It's about respecting the cognitive load of the person using the tool. Every unnecessary abstraction, every verbose method name, every surprising default is a tax on human attention. The best code is code that doesn't make you think about the code.

Error philosophy. How a library fails tells you as much as how it succeeds. Do you raise custom exceptions or let Python's built-in errors propagate? Do your error messages blame the user or guide them? Do you fail fast or try to recover? These decisions are deeply personal, and they pervade every function in the codebase. I wrote about this in the context of designing for the worst day — the error experience is the experience for someone having a bad day.

What you leave out. The features you don't build are as important as the ones you do. An HTTP library without a built-in caching layer. A music library that doesn't try to model rhythm or tempo. These omissions create a clear boundary around what the library is, and AI can read those boundaries. When AI suggests a new feature, it rarely suggests something that violates the scope you've implicitly defined — because the scope is legible in what's already there.

```
# What V1 encodes isn't just code.
# It's a philosophy, compressed into working software.

class Library:
    """Built by hand. Extended by AI."""

    def __init__(self):
        self.architecture = "human instinct"
        self.naming = "for humans"
        self.errors = "compassionate"
        self.scope = "deliberately bounded"

    def extend(self, feature):
        """AI reads the existing patterns
        and follows them faithfully.

        The foundation does the teaching.
        """
        return self._match_existing_voice(feature)

    def _match_existing_voice(self, feature):
        # This is what a hand-written V1 makes possible.
        # Without it, there's nothing to match.
        ...
```

The Problem with AI-Generated V1

I've seen the alternative. I've experimented with letting AI generate initial versions of small tools and utilities. The code works. It's competent. It follows best practices. And it has no personality.

AI-generated V1 code tends to be correct but generic. It uses conventional patterns because it's been trained on millions of repositories. It names things sensibly but not memorably. It structures things reasonably but not opinionatedly. It's the architectural equivalent of a hotel room — functional, clean, forgettable.

The deeper problem is what happens next. When you ask AI to extend AI-generated code, it has no strong voice to follow. So it generates more generic code. Each iteration is reasonable on its own, but the cumulative effect is a codebase that feels assembled rather than authored. No one's taste is embedded in it. No one's philosophy shapes it. It's software by committee, where every member of the committee is the same statistical average of all code ever written.

> This is a specific instance of a general problem with generative AI: without strong constraints, it regresses toward the mean of its training data. A hand-written V1 provides those constraints. It's like giving a jazz musician a melody to improvise on versus asking them to improvise with no theme at all.

When I write V1 by hand and then let AI extend it, something different happens. The AI treats my code as the authority on how this particular project should work. My naming conventions become its naming conventions. My architectural patterns become its architectural patterns. My error philosophy becomes its error philosophy. The AI becomes a faithful collaborator rather than an independent author, and the result is software that feels like one person built it — because one person did build the foundation, and everything else followed from that foundation.

What This Looks Like in Practice

In my experience, when a library has a strong hand-written foundation, AI follows it instinctively. The naming conventions, the module structure, the way errors are surfaced — it reads all of it and matches the voice. New features feel like they belong because there's a clear voice to belong to.

When the foundation is weak or generic, the opposite happens. Each AI-generated addition drifts slightly from the last. There's no center of gravity. The codebase accretes rather than grows.

The pattern is consistent: human writes the foundation, AI extends it faithfully. The human provides the taste. The AI provides the throughput.

Why This Connects to Everything

This is the recursive loop in a new form.

I've written about how programmer consciousness shapes code, which shapes user consciousness, which shapes collective consciousness. The "write it first" principle adds a new dimension: programmer consciousness shapes the initial codebase, which shapes how AI extends the codebase, which shapes what millions of users experience.

If you write V1 of a library with care — with a genuine philosophy embedded in every API decision — then AI's extensions carry that philosophy forward. Your values propagate through code you didn't write, because the code you did write taught the AI what values to propagate.

> This is why the quality of V1 matters so much. It's not just the foundation of the codebase. It's the template for everything that comes after. A carelessly written V1 propagates carelessness. A thoughtfully written V1 propagates thoughtfulness. The AI is an amplifier, and amplifiers don't add taste — they amplify whatever signal they receive.

This connects to programming as spiritual practice too. The meditative care you bring to the initial design — sitting with a naming decision for an hour, restructuring a module three times until it feels right, deleting a feature because it doesn't serve the core purpose — that care doesn't just produce better V1 code. It produces a better teaching artifact for AI to learn from. Your spiritual practice in writing code becomes encoded in AI's ability to extend that code with the same spirit.

The hand-written foundation is an act of consciousness. Everything that follows from it inherits that consciousness, whether it's written by you, by AI, or by a contributor who reads the codebase and absorbs its patterns.

The Deeper Principle

There's a principle in here that extends beyond code.

In any creative collaboration — whether with AI, with other people, or with future versions of yourself — the quality of the initial vision determines the quality of everything that follows. A strong opening chapter gives a ghostwriter something to match. A clear architectural sketch gives a builder something to realize. A well-designed pilot episode gives a writers' room something to extend.

The initial creative act is where human judgment is irreplaceable. Not because AI can't generate plausible first drafts — it absolutely can. But because a first draft without embedded taste gives collaborators nothing to learn from. The first version isn't just the starting point. It's the style guide, the architectural pattern, the voice memo, and the philosophical statement all in one. It's how you tell your collaborators, human or artificial, what kind of thing this is.

Write it first. Make it yours. Pour your experience and your instincts and your years of hard-won taste into that initial version. Make it opinionated. Make it personal. Make it the kind of code that someone could read and know you wrote it.

Then let AI drive. Let it add the features you've been meaning to get to. Let it write the tests that are tedious but necessary. Let it clean up the edge cases and improve the documentation and handle the maintenance burden that would otherwise drain your creative energy. Let it do all of this while faithfully extending the voice you established.

The human provides the soul. The AI provides the scale. That's the partnership.

```
# The collaboration model that actually works.

class Project:
    """Human-initiated, AI-extended."""

    def __init__(self, creator):
        self.voice = creator.taste
        self.architecture = creator.instincts
        self.philosophy = creator.values
        # V1: written by hand, with care.

    def extend(self, ai, feature):
        """AI reads the voice and follows it.

        Not replacing the creator.
        Amplifying the creator.
        """
        return ai.implement(
            feature,
            style_guide=self,  # The codebase itself.
        )
```

This is what the maintainer as interface looks like in the age of AI collaboration. The maintainer's role shifts from writing every line to establishing the voice that every line follows. The interface isn't the code anymore. It's the consciousness behind the code — the taste, the philosophy, the carefully considered opinions that make a library feel like it was built by someone who cares.

You can't delegate that. You can only amplify it.

Write it first. Then let AI drive.

For more on the philosophy behind this approach, see Programming as Spiritual Practice and The Recursive Loop.