

Modern Software Engineering with AI

Governing Agents in the Software Development Lifecycle

Antti Koivisto

Copyright © Antti Koivisto, 2026. All rights reserved. No part of this book may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical reviews and certain other noncommercial uses permitted by copyright law.

AI coding tools promise dramatic productivity gains, but getting beyond demo quality requires more than better prompts. This book covers what it actually takes to deliver reliable software with AI agents: the governance, the skill shifts, and the hard lessons from real projects.

You may read it for free and without any forms to fill.

If you find it useful, I ask you to donate a sum you can afford to a charity of your choice.

Contents

- Foreword 4
- Introduction 8
- From Autocomplete to Software Factory 14
- The Cognitive Flood 25
- Why This Is Harder Than It Looks 36
- Everyone Wants a Software Factory 42
- The Compound Probability Problem 47
- From Writing Code to Directing Agents 57
- The Skills That Don't Come from Coding 67
- Why Organizations Resist (and How They Adapt) 77
- Accountability Over Speed 86
- The Pipeline: Stages, Gates, and Artifacts 90
- Architecture as Probability Management 100
- The Framework in Practice 109
- The Specification Premium 132
- Developer Intervention Required 141
- Where the Framework Runs Out 148
- The Beef and the Boilerplate 158
- Context Engineering in Practice 179
- The AI Tool Ecosystem 198
- Modernizing Legacy Systems 206
- Closing the Feedback Loop 217

Foreword

Most of you have probably seen the hype, promises and speculation about AI-assisted development. Countless social media and marketing materials speculate about the future of our craft and even employment, with so far largely unfounded claims about productivity gains.

Demos of complex applications built in minutes. Conference stages feature engineers describing how they shipped features that once took weeks in a single afternoon. People on social media telling how many tens of thousands of lines of code they have generated with AI.

Many of them have a bridge to sell? You bet. My experience, shared by many of my colleagues, has made me quite sceptical about the boldest statements. Yes, wonderful tools, but still hopelessly unpredictable and unreliable unless you keep a very close eye on them or you're just after quick wins. Larger scale, reliable, maintainable software? That's a different story.

There was no single moment of failure that made me write this book. It was more of a progression. As our ambition level grew beyond "fix this" tasks, the standard off-the-shelf features started to fall short in a team setting. Agents tried too hard or gave up too easily, made shortcuts, or produced uneven output. The single instruction files, MCP servers, and bolt-on tools added to confusion rather than reducing it, and at times it felt like things got worse rather than better. What I gradually realized was that the gap between "impressive demo" and "reliable delivery" is not a tooling problem. It's a governance problem.

It's more complicated than that. AI-assisted development can indeed make you dramatically more productive, but it is not a magic wand some seem to believe. Getting the right things right requires a lot of hard work, and the organizational challenges are significant. In some cases they might even outweigh the benefits of the AI-powered approach.

The basic question I set out to answer with this book is: what does it take to get your AI-generated app beyond demo/PoC quality in a sustainable way?

Who this book is for

I wrote this for the people who have to make AI-assisted development work in practice: engineering leaders, delivery leads managing scope and timelines, architects concerned with system integrity,

and senior developers who understand that tools are only as good as the processes surrounding them.

If you are looking for a tutorial on how to use Cursor, Copilot, Claude Code, or any other specific tool, this is not that book. The tools change too quickly, and the real challenges are not in the tools themselves but in how we organize around them. I won't be handing out specific prompts that will make AI generate perfect code. I won't be making you more assured that AI will solve your software delivery problems.



I hope you'll find a practical starting point for structuring teams, processes, and accountability when code becomes cheap to produce but expensive to verify.

Why this book, why now

There has never been a better time to write this book and to start taking things into practice. Looking back my 40+ years of programming with all hypes, new things coming and going, I have never seen a technology with such a potential to change how we work. If somebody had told me 30 years ago that one day we would tell the computer to write very complex software for us and it would do it, I would have thought that was pure science fiction. And yet here we are.

This book was inspired by my previous work on the topic, including a book together with an amazing co-author, JP Heimonen.

Another motivation were the needs of a real project on the horizon and the long-forgotten draft table of contents I had pieced together. For the real application, and considering myself relatively well informed on this topic, I realized we need to build something on top of the tools to make them work for us in a sustainable way. The standard ones, even the best ones, were frankly not good enough for the kind of work we needed to do. How come they are still stuck in a single developer, single session paradigm with naive approaches to context management? None of this is gonna work in bigger setups.

➤ [Handbook for AI-Powered Developers](#) — Handbook for AI-Powered Developers

This book, however, has been my personal ambition.

I've used the AI intensively for pretty much everything ever since the beginning. I think that only in the last few months both models and tooling have reached a state of being good enough for more ambitious applications, and I certainly feel that the time is right to share the insights and practical guidance I've gathered along the way to help colleagues on track with this with a realistic and responsible approach.

Hope you find some useful insights in the book.

And for those who know me, expect whatever you find inside to be littered with (bad) jokes, complicated sentences and a healthy dose of skepticism.

About the author and acknowledgements

I started programming over 40 years ago and have been doing it professionally for almost 30. I've been an architect, a developer, a team lead, and occasionally all three at once. I've survived waterfall, survived Agile, survived SAFe, and now I'm trying to survive AI. Along the way I've worked on everything from tiny startups to large enterprise systems, and I've collected enough scar tissue to fill a book. This one, apparently. Much of what's in here I learned from colleagues and customers over the years. They are far too many to name, but you know who you are.

i I work at Siili Solutions, where I build software for clients. This book is my private venture, and the opinions expressed here are my own.

Throughout, I have tried not to oversell what I think is possible right now. AI-assisted development is moving fast, and some of what I write here will need revision as the field evolves. But I argue that the underlying principles about accountability, traceability, and the necessary boundaries between your judgment and automation will last.

Many thanks to all colleagues and customers along the way for their patience.

Tornio, Finland March 2026 Antti Koivisto

Want to reach out? Find me on LinkedIn or send an email.

E-mail: anttikoi@gmail.com LinkedIn: [linkedin.com/in/antti-koivisto-24646a](https://www.linkedin.com/in/antti-koivisto-24646a)

P.S. In case you disagree on my points, or think they're a load of BS, I'll respond by quoting a former politician. There was a rumour of him saying, 'I once thought that I was wrong. But I was mistaken.' So bit like him, I'm gonna stand by my work but reserve my right to be wrong.

i **Disclaimer and notes on AI usage as a writing assistant**

This book is based on my personal experience and observations. While I have tried to be as evidence-based as possible, it is not a complete or definitive guide, a scientific study. It's also not marketing material.

So take everything as rather opinionated and subjective with healthy doses of salt.

All content and text is written originally by me, but proofread, reviewed, reorganized and edited with the help of AI tools. All the images have been created as code (svg) with AI (thought sometimes it might have been easier to do it the 'old way')

Introduction

This book focuses on the application of generative AI for software engineering. While some of the ideas I'm going to discuss are specific to my field, some of them apply to other applications of generative AI as well. The core principles of accountability, traceability, and governance are relevant whenever we are using probabilistic tools to produce critical outputs.

Creating or modifying computer software has never been this **quick, easy or fun**. So I've added a backstory (or a sidetrack) for this book about a long-postponed hobby project, a roguelike game clone. I did it entirely using generative AI and freely available tools. Just a couple of years ago I wouldn't have even started because of the effort and time required. This is not supposed to be a **wow factor** here, or the "look how many lines of code you produce in a week" demo, but just a little showcase of how much you can do with the tools available today in a short period of time.



You'll recognize my take-home lessons from this little project from the following formatting.

@ I'd always just wanted to add ranged attacks with aiming to Nethack, but never had the time for that. But rot.js, Opencode and also this little book project gave me some motivation.

Let's dig a bit deeper.

Free code

As the price of code goes down, the expensive parts of software delivery become more visible. Right?

Think coordinating multiple developers working on the same system. Or ensuring that generated code actually solves the right problem and does not invent any non-existing ones in place or introduce new ones. And doing this while maintaining quality standards when the volume of things to verify and check goes through the roof.

A rigorous study by **METR (2025)** found that experienced open-source developers using AI tools actually took 19% *longer* to complete tasks. What was noteworthy was that they had predicted a 24% speedup beforehand, and still believed AI had helped afterward despite the opposite being true. The gap between perception and reality is significant, and [Chapter 3](#) examines this and other evidence in detail.

➔ [METR](#) — Early 2025 AI-Experienced Open-Source Developer Study

Google's DORA reports tell a nuanced story at organizational scale: the 2024 data showed AI adoption correlating with *lower* throughput, while the 2025 sequel found that penalty had flipped, though delivery instability still goes up. [Chapter 3](#) examines this and other evidence in detail.

Within the past year we have much better capabilities and emerging disciplines at our disposal that allow us to raise our abstraction level from the code to the requirement. I'm about to discuss this transition, often referred to (much to my dismay) as **Specification-Driven Development (SDD)** later in this book. It's essentially about moving the abstraction level from code to requirements, planning and verification.

The gap between perception and reality matters. Organizations are making strategic decisions about AI adoption based on impressions that may not match the facts on the ground, which might lead to disappointment and disillusionment when the expected productivity gains do not materialize. One of the goals of this book is about avoiding that trap.

In the software bubble the situation on the ground is still like **after the first hot contact with the enemy**. Many are restructuring workflows around tools that may be creating coordination overhead faster than they reduce implementation time. I might very well be one of them. Engineers are developing habits—like deferring to AI suggestions, skimming generated code—that may degrade quality over time, and for sure, they don't have the same grip as they would in the old world.

The bottom line here is that we're now at a point where **individual productivity gains do not automatically translate to team productivity gains**. Unless we do something about it or wait for somebody to create that magical software factory. You know the one which will just churn out (basically brute force) your app/solution/whatever with little to no involvement from you.

i The evidence for AI coding productivity is conflicting. My take on this is that the initial speed and lots of printout seems good at first, but end cleanup and troubleshooting in the end tends to take equal or more time than the traditional debug-fix-test cycle.

The evidence gap

Beyond the METR study cited above, the evidence for AI coding productivity is mixed and methodologically weak. Many studies measure tasks that are not representative of real software development work, use toy problems, short time horizons, or metrics like lines of code that do not correlate with delivered value. Perhaps unsurprisingly, studies sponsored by AI vendors tend to show larger benefits than independent research.

This does not mean AI tools are worthless or cannot be astonishingly useful. Many of them (and I've used quite a few since the early days) are impressive, and many developers find them genuinely helpful. But it does mean we should be skeptical of claims about 10x productivity or revolutionary transformation. The evidence does not support such claims, and organizations that bet heavily on them may be disappointed.

The probabilistic challenge

Large Language Models (LLMs) are probabilistic systems. They generate outputs based on statistical patterns in their training data, not on logical deduction or formal verification. This means they can produce excellent code, terrible code, or subtly wrong code that looks excellent on first inspection. Same prompt with exact same sources might yield totally different results on the next run.

i Software engineering, by contrast, needs determinism. A given input should produce the same output. Edge cases should be handled consistently. Behavior should be predictable and verifiable. A good rule of thumb is that any deduction, sequence or data processing you can easily do the old way should be done that way. Just use GenAI to develop that solution.

This conflict is fundamental, not incidental. The probabilistic nature of LLMs is not a bug to be fixed, and we won't get *entirely* deterministic LLMs. What we can do is design processes that manage the probabilistic output in ways that produce reliable outcomes. This is a major theme of this book.

Chapter 5 explores this challenge in depth: the step size principle, probability management through specification, and why smaller, bounded tasks produce more reliable results than open-ended generation. The core insight is that we are not trying to eliminate probabilistic variation—we are trying to constrain it to spaces where variation produces acceptable outcomes.

The response

What I'm going to present in this book is a practical, learnable approach to AI-assisted software delivery. I try to address issues like coordination, correctness, and accountability. The principles of my 'framework' (note the quotes, this is not TOGAF, Scrum or SAFe 3.0) rest on three simple controls to tame the probabilism and create a reliable delivery process, or set the foundation for a 'software factory'.

Mechanism	What It Does
Gates	Controls which agent is allowed to work next, and on what. Enforced transitions between stages prevent wandering and keep things focused and traceable.
Tracking artifacts	Record what was planned, built, tested, and approved. Machine-readable specifications tied to specific backlog items, providing the single source of truth.
Human checkpoints	Hard stops in the state machine. Validate understanding before code generation, validate implementation before delivery, and retain authority over what enters production.

Together, these mechanisms create accountability without sacrificing the speed that makes AI tools valuable. Let the AI work freely inside gates and get involved to make the decisions that matter. The result is something you can trace, review, and actually explain to someone, even when most of the code is generated by AI.

How this book is organized

Part I lays out the problem space: the cognitive overload facing teams, what the research actually shows, the field of "software factory" attempts, and the compound probability problem that makes multi-agent pipelines unreliable without structure.

Part II addresses the human side: how developer roles change when AI writes the code, what new skills matter, and why organizations resist new ways of working.

Part III presents the governed delivery approach: accountability as the goal, the pipeline mechanics, architecture as probability management, SDD, and where you need to stay involved.

Part IV is the practitioner's field guide: open problems, getting started, context engineering, tool selection, legacy modernization, and closing the feedback loop.

The book closes with a look at where software engineering is heading and what this moment means for our profession.

This book has been designed, perhaps optimistically, to be read from beginning to end, but feel free to jump around. The core principles are introduced early and then applied throughout, so you can

pick the sections most relevant to your current challenges. The field guide in Part IV is meant to be a practical reference for implementation details.

Your reading companion: AI Bullshit Bingo

Before we begin, here's a little interactive companion for your next AI strategy meeting, keynote, or LinkedIn doom-scroll. You might find it also entertaining while reading this book. TIP: If you're reading the web version of the book, just open a new tab for it!

I prepared this especially for my wonderful but oddly Bingo-loving colleagues at Siili!

B	I	N	G	O
"10x developer"	Agentic workflow	"AI-native company"	Vibe coding	AGI by next year
"Software factory"	AI replaces developers by 2026	Spec-driven development	Prompt engineering is the new coding	Multi-agent orchestration
"It wrote the whole thing"	Copilot for everything	★ FREE ★	"Ship 10x faster"	"It just hallucinated"
AI slop	AI-powered [noun]	"AI-first"	Thought leader	"The last developer"
Autonomous agents	"Reasoning model"	Trough of Disillusionment	Superintelligence	"We'll just use AI for that"

Click the squares as you encounter them in the wild. Free space is on the house.

1

From Autocomplete to Software Factory

Levels of AI assistance and what each demands

How do we use AI? Are we using it at the basic level, perhaps as a detached research tool (Google/StackOverflow replacement)? Do we use it for editing or explaining selected sections? To analyze or refactor within a single file or across multiple? Do we let the toolset build the plan, or do we strictly micromanage each step?

This chapter uses a ladder metaphor to illustrate the progress from simple, atomic tasks to complex, fully autonomous software factories. Each of these types has different implications for how we structure our teams, processes, and governance, and how much guidance, up-front planning, and supervision is required to get the most out of them.

For me it's usually a starting point to approach new topics through a Venn diagram, layer model, or something along those lines. Once an architect, always an architect, right? Anyway, I think there's a clear 'maturity model' or 'ambition level' about how much we're willing and capable of letting the AI do for us. You could argue that the trend is just an additional abstraction on top of old; bit like from Assembly to C to Java to Python. Or, from proprietary system to no code/low code or commercial SaaS services or platforms instead.

Over the course of my career, I've been involved hands-on in the actual development since the beginning and to this date. Sometimes to the annoyance of many developers, no doubt. Over these years, I've become less and less concerned about code: what language and stack you use is not as interesting as it was just a few years back. What's important is what it does and to what end, and of course, that the guy maintaining it won't come after us somewhere in the future. Fact is that most issues you encounter in typical software projects have ceased to be technical problems at all a long time ago, but organizational and human problems. The same applies to AI-assisted development: making the most of it means you need to get out of the 'code zone' regardless of how much pain that causes, and start thinking more about deliverables, processes, and workflows.

Four Levels of AI Assistance

From autocomplete to autonomous decision-making



In the figure above, the starting point is something like autocomplete++. At the end of the spectrum (rainbow?) we have the fully autonomous AGI software factory. The mass-production of software is supposedly something where we just feed in high-level requirements and the system churns out a working product with minimal to no human involvement.

i The level 3, the Factory 0.1 is definitely within reach or a reality for increasing part of the developer population. What it takes to get there and beyond is the thing I'm trying to shed some light with this book.

Developer perspective: The levels of AI assistance

The ladder model above is probably not going to convince anybody in the frontline trenches and foxholes. I know this because that's where I spend most of my working hours, too, so I figured to approach this from more practical perspectives.

Going up these ladders, developers are gonna ask questions like:

- What do you really need to do to make this work?
- What is the scope of the task on each ladder?

- How much context do you need to provide to the AI to get good results (as a function of task size)?
- What kind of instructions and specifications do you need to give?
- How much review and iteration (and rework, rewrites, debugging, ...) is typically required?

Figuring out the starting point is of course a good place to start. There aren't many developers out there who are not already at least halfway on board. According to a recent poll on social media by a CEO of a Finnish company Sysart, Petri Mäenpää got 300 answers to a question: to what extent do you use AI tools in software development. The results were pretty clear: only 11% said they don't use AI tools at all, 36% were on the agentic level, and 24% were not writing any code themselves at all.

➤ [Petri Mäenpää's poll on LinkedIn on the level of AI tools in software development](#)
[17.02.2026](#)

I'll walk through these levels below and will revisit them several times later in this book as well.

Atomic: Small context and easy to verify

The most common use of AI tools is rather simple. Ask for a specific thing related to a small context, and get a specific answer. This was the original offering equal to the browser-based AI prompting that people often refer to as 'AI'.

Goal	Fix a specific issue or make a small change with minimal context.
Context size	1-100ish lines, or a single error message.
Interaction style	Developer chooses what to include, and applies the changes manually to the source.

Imagine editing a single line, figuring out a syntax error or a linter warning, the context and intent are clear: fix something, change something, explain something. For example:

▶ Example

➤ **You:** "Make this function async and add error handling."

◆ **AI:** "Here's the modified function with async/await and a try/catch block."

This kind of tasks are quite straightforward, and with good project configuration, results are consistently good. Bit like Intellisense on steroids. This kind of ability has been remarkably well integrated into most IDEs already for some time, so let's move on to a more ambitious level.

Smart: Single file context, refactoring, and feature additions

When we aim for a bigger scope, often hoping to save some effort, the problem becomes more complex.

Goal	Refactor or add functionality to a single file, module or a page.
Context size	A few hundred lines, or a single file.
Context source	User selects the file or section to edit.
Interaction style	Developer chooses what to include, and usually applies the changes one-by-one.

So when operating on this 'smart' level, you basically delegate tasks which span over a single module (a single file or a handful of files), the scope widens perhaps to a few hundred lines in total with a handful of cross-references to worry about.

This is pretty much the "Edit" mode in the early AI code assistant tools, where you could select a block of a file, and ask something to be done with it. The AI didn't have to look for more advice (and it didn't) and the new capabilities were more related to for instance refactoring, adding new features, or other reorganizing still keeping in a relatively small scope.

So let's consider a practical example.

► Example

> **You:** "Add a new function to calculate the average of an array."

◆ **AI:** "Here's the new function added to your file." (+20 new lines)

This level of AI-assistance provides better results than what I can write myself, with a lot better comments. For the developer, the amount to check and verify typically remains manageable, and the risk of introducing new bugs or issues is low (or not any greater than you doing the same manually). And it's easy to revert if the damned thing doesn't compile after all or breaks a test.

✓ IDE-integrated editing tools have been around forever. For instance, the LSP (Language Server Protocol) powered refactoring tools are still much faster for simple tasks like adding a parameter to a method, extracting a method, or renaming something. And won't eat your premium token budget. No need to use the AI for everything when you know the tricks.

When keeping our expectations on this level, the results are astoundingly good. Yes, you're likely to encounter hallucinations, usually in the form of assuming something that is not there. Like a non-existent method in another class being called, or adding some parameters to a common API which you should not touch. These are usually easy to spot and fix, as they are isolated incidents. You might be guessing where I am getting here when we go further: what if there were 100 places to fix after your new feature or fix spanning a lot of files or modules? Well... that's what you're gonna see happening when your elevator takes you up to more advanced levels. Not that AI wouldn't be able to fix the issues by itself -- it often is, especially if the error is something a compiler or linter can catch.

Agentic: Multiple file / system context working semi-autonomously

This is the level the majority of developers are already at, at least in some form. Compared to the previous level, the difference is huge. You merely express intent -- 'Do this' -- and let AI figure out what needs to be done and where, perhaps with some tips for the starting point, such as a stack trace or a test failure.

Goal	Implement a feature that requires changes across multiple files or modules, or refactor a subsystem.
Context size	Thousands of lines, or an entire codebase.
Context source	System discovers relevant files based on the task, or user selects multiple files.
Interaction style	Developer provides high-level instructions, and the AI determines which files to modify and how.

When scope expands to multiple files, regardless of whether they are selected by the user or discovered by the system, the problem becomes significantly more complex. Needless to say, the probability of misinterpretation increases substantially. Let's compare this to early 'Agent' modes in AI code assistants, where the tool could access the entire codebase and make changes across files.

This is also the level where 'direct' hallucinations, like made-up functions and naming errors, are most common, and where the early adopters often got most frustrated. Trickier problems arise from conflicting changes to several files, which for instance failed to identify the correct sequence of things to be done, or failed to understand e.g. intricacies of your front-end stack's page lifecycle. In the past couple of years, however, this kind of task is in much better shape, especially if your 'design patterns' are properly referred to in the context (more about that later). Many of the *subtle bugs* arise from this: syntactically correct code that looks very good does not work as intended.

Let's consider a practical example. This might work right off the bat, as it is probably something repeated in thousands of examples in the modern LLM's training data (might use an outdated API though).

► Example

- > **You:** "Refactor the authentication module to support Google authentication."
- ◆ **AI:** "Here's the refactored module with Google authentication support."
(10 changed files, -1000 lines, +2000 new lines)

Anyway, ambitious refactoring across files, adding new features that require changes in multiple places, or even implementing a new module with interactions across the system are well within the capabilities of rather off-the-shelf configured agent-based tools, especially when you keep your ambition level in check and still know pretty well what you are doing.

⚠ Also with AI fighting on your side, you need to be able to validate the answers you get. Like "I know the correct answer but I'd happily let you do the heavy lifting to get there."
If you can't, you're no longer practicing engineering.

Key difference between 'Agentic' and the 'Smart' level is the context management and the iterative nature of things getting done. The AI figures out what it needs, and often tries to do so until it hits a barrier and/or your premium token limit. The developer no longer explicitly tells it what to look at, but relegates the 'blast radius estimation' and the dependency management to the AI. And as easy as it sounds, *working* code is not just text that you can traverse, but a dynamic, structured beast hard

enough for a human to understand. There's a reason why we have debuggers and good Intellisense features.

Let me give you an example where things easily go south when operating on this level. Let's imagine you want to add new properties and behaviors to existing code, let's say some controls in your React app. Unfortunately, as it often is, this kind of technology may seem rather easy, but you need to know the higher-level patterns, possible limitations, and good practices to get it right. Just typing syntactically correct code, which passes the linter and even your unit tests, does not prove ****.

I've never really liked dynamically typed programming languages like JS or Python. Just too easy to do things seemingly right and hit an error due to a syntax error, missing dependency, missing property or whatnot. And only discover that at runtime. It's bit like that with AI

For instance, forgetting about proper cleanup, or using things that the linter does not catch but cause runtime errors, still happens.

 When using agents, the amount of changes per session easily becomes too much to carefully review. Yes, I admit just pressing that 'Keep' or 'Accept all Changes' quite a few times, and yes, have often regretted that. "Commit often" is an essential practice to follow here: be prepared to roll back even long sessions (and why not, it's not your sweat and blood that paid for it).

Next, let's look at what happens when we start using much more independent agents to do the work for us.

Multi-agentic: System/product-level context

Goal	High-level feature development, architectural changes, and major refactoring.
Context size	Entire codebase, documentation, and relevant external information.
Context source	System discovers relevant context: code, documentation, and external resources.
Interaction style	Developer provides high-level requirements as documents and prompts.

In a way this is the Factory 0.1 (or perhaps, Factory 0.01) level as suggested in the ladder model. Instead of typing, refactoring, and refining the code, the developer is reduced to describing their input in documents or a multi-stage interview instead of prompting something simple (certainly that can still be done as long as the intent is crystal clear). This intent may be detailed or very simple based on the task at hand, but based on that, the 'factory' is expected to autonomously determine the necessary context (what to touch), what actions to take, and what changes need to be made, and somehow verify the outcome.

► **Example**

- › **You:** "Implement a new recommendation engine based on user behavior data."
... a lot of tokens and time are spent ...
- ◆ **AI:** "Here's the new recommendation engine implemented across the system." (Changes in data processing, database schema, API endpoints, and frontend components) (+1000 new lines, 20 changed files, +10 md new files with 2000 lines)

A major theme in this book is really about how to make this level of work both technically sustainable and organizationally manageable. And how the developer can remain in control and stay sane.

i Even with the sample numbers thrown in above (which are easily on the low side when you get serious), additional governance and checkpoints are needed, no matter how competent the coding model you have is. At this scale AI **will** get things sometimes horribly wrong, in ways that are hard to detect. Nobody is going to or even able to review all that. That's the case without AI already: if you think somebody will review your 1000-line PR properly, you're fooling yourself.

Then onto what this means from the organizational perspective.

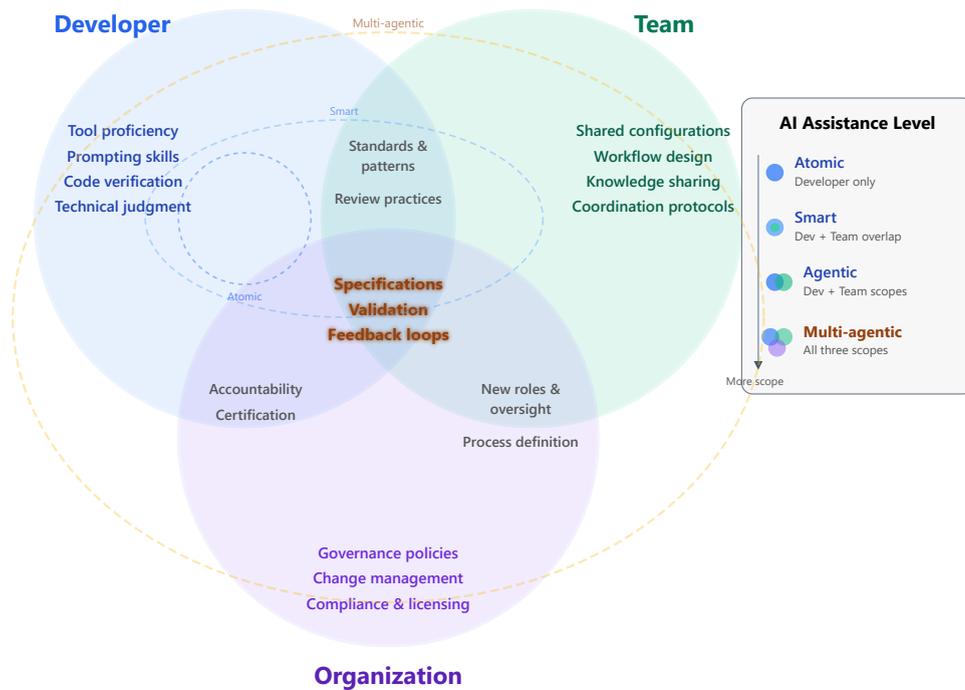
Business scale and scope

Final thing I'm going to briefly touch here is the business scale and scope of my ladder model. Using a T-shirt-like challenge sizing model (Scope), we see a (rather obvious perhaps?) pattern. The bigger the challenge or scope, and the more you look forward to getting from AI, the more you need to up your game in terms of AI adoption and the legwork around it.

I tried, engineer as I am, to collect the above into a Venn-ish diagram to illustrate the relationship between the levels of AI assistance and what kind of things you need to have in place to get there at different scopes.

Scaling AI Adoption: Who Needs to Change?

Higher ambition requires alignment across all three scopes



The three scopes of AI adoption and how the levels of AI assistance expand through them.

In case you're not exhausted (or, bored) by this onslaught of complex diagrams, check the table below. It tries to summarize the relationship between the scope of the problem you're trying to solve, the level of AI assistance you need to get there, and what kind of organizational changes you need to make to get there. Yes, this is indeed a gross simplification of a very multivariate problem, but simply, **the more ambitious your goals are, the more you'll need to up your game** in terms of AI adoption and the legwork around it. In short, I'll *postulate* that no amount of jargon, you know, 'spec driven development', 'agents/agentic', 'factory', or buying more licences (and, pay-as-you-go tokens) of the next best-thing-since-sliced-bread CLI coding agent tool thrown on PowerPoint is gonna get you there. You need to learn to use them properly, and you need to have the right processes, governance, and organizational structure in place to get there.

Business Scope	Single Developer	Team	Company
Small	Atomic	No change	Buy the Copilot. Get approval from the customer.

Business Scope	Single Developer	Team	Company
Feature	Smart	Shared settings and configurations	Training, certificates
Stream	Agentic	Build the workflow around AI assistance	Processes, oversight, new roles
Product	Multi-agentic	Build the coordination layer and tooling	Establish governance, train and nominate experts, manage change

So, we've concluded (my snide remarks about slideware aside) that to reach the multi-agentic 'Software Factory' churning out production-ready code with minimal human involvement 24x7, a lot needs to change. You need to up your game from the developer all the way to how the company operates. Again, there's no product that does this with just a few simple prompts for anything more complex than the simplest apps. In the wild, things get more complex, restrictive, are often subject to regulation, have portions with no counterparts in GitHub (and hence, are not something your models know already). Coding that simple web page for a toy problem, or generating an entire website, does not prove a thing. You could clone a similar one from GitHub in minutes with no LLM any time.

! The more you expect and the more ambitious you are, the more you need beyond Copilot licences and 2 hours of training for developers.

- A single developer working on a feature can use less formal processes.
- A company-wide product change requires extensive specification, validation, and coordination across the full system and teams.

Why this matters

If you are to get more ambitious, I'd suggest to get the following in order. Limit the creativity, validate everything at every step, have clear rules (also beyond the agentic world), and align your team and the methods together.

Perhaps this is the 2026 version of the Enterprise Elevator, which used to take us architects from the basement to the C-suite?

- ? **Better specifications** Constrain the search space
- ? **Stronger validation** Catch errors before they become catastrophes
- ? **Clear governance** Ensure accountability
- ? **Team alignment** Make the system predictable

Things I list above sure aren't new or even really specific to AI. In the old days you could fend your way out even if everything wasn't that systematic, governed etc.

! With traditional development, a muddy specification might slow you down. With AI-assisted development, a muddy specification sends you in the wrong direction entirely.

2

The Cognitive Flood

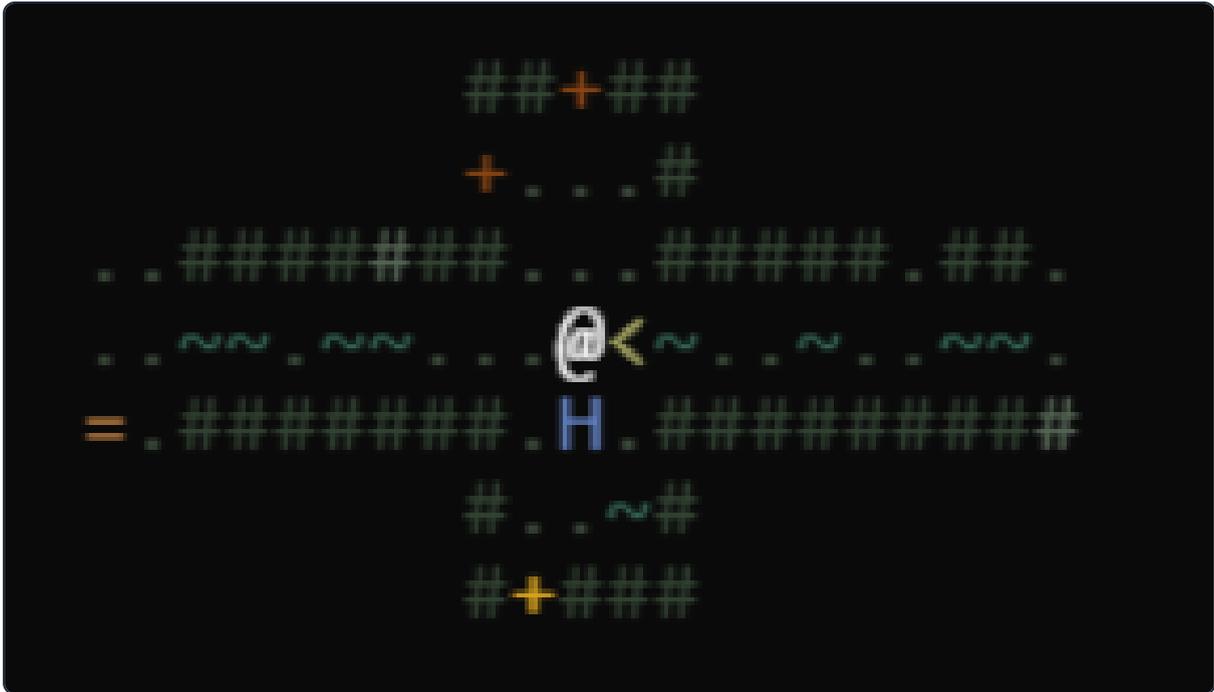
What a developer faces when entering the AI-assisted development world

A I coding tools are extraordinarily powerful and getting better all the time. Developer-oriented solutions like Claude Code, GitHub Copilot, Cursor, Windsurf, Antigravity have slightly different approaches but seem to rapidly reach feature parity. Similarly, more product-oriented services, such as Lovable, seem to produce more and more finished solutions with just a single prompt, and gain more and more features every month.

They all suffer, regardless of the latest context engineering feature or a UX trick, from similar issues when applied without proper guidance and expertise. Examples of these symptoms, but not limited to, are wandering agents, rotten contexts, and results that can't be traced back to the requirement. Or, total misses like the product contains pretty much everything else but the thing you really wanted, and even those often barely working.

Worse yet, the larger the codebase gets the worse the results get. Throwing more tokens, iterations or subagents may help sometimes but they are essentially brute-force techniques which are not only expensive but very ineffective.

Let's take a moment to look at what happened to my game in the beginning.



GSD + Opencode: early roguelike project setup

@ When starting my roguelike project, I was very well on board with the domain. I picked Get-Shit-Done (GSD) as my project manager, planning assistant, and agentic coordinator as I felt it is surely enough for a single-developer problem. So it was, nicely integrated with Opencode, a real quickstart and easy to interact with. The biggest issue was perhaps to understand the assumed project structure of milestones, roadmap, and phases, each of which have the same lifespan. (Btw I've grown much more fond of Opencode than Claude Code).

All these new techniques and tools are powerful, but add to something I call **concept overload**. Some might refer to it as the "cognitive load", but we're talking about essentially the same thing and I've probably used these terms interchangeably in this book already. Think about if your workplace introduced new processes, new paradigms, new strategies, new tools so frequently that it's impossible to keep up. That's the situation for many right now.

The issue isn't just a cognitive problem, but an emotional one. Concept overload is causing anxiety for people whose careers were built on skills that are now being automated. The developer who spent a decade mastering a framework watches an AI generate equivalent code in seconds. The architect who prided themselves on system design sees AI proposing (better) alternatives (I'd be the first to admit it). The uncertainty isn't abstract; it's personal, and it colors every interaction with the new tooling.

Introduction to new concepts

Consider the following image: how many totally new concepts are there for an average developer to learn when they start AI-assisted development? A lot. And this is just the beginning: not only do you need to master these terms, you also need to become 'AI-literate' and learn how to avoid getting 'AI-lazy'.

The Concept Explosion

Five dimensions of emerging complexity in AI-assisted development



Let's split the concepts used in the diagram above into six categories. (The diagram btw, as all the others was indeed generated by my trustful AI companion as SVG/JSX in case you're interested. None of them were, regrettably, exactly one-shotted)

Processes	Methods like prompt engineering, spec-driven development, vibe coding, software factory patterns are all ways to organize work and interact with the AI.
Agents	Non-determinism, orchestration, swarms and parallelism, function and tool calls, skills (to name a few) are the mechanics of how AI agents actually operate.
Models	Classes and sizes, providers, token usage, mixture of experts, chain of thought, thinking modes, system prompts, custom instructions are vague terms related to LLM models and their usage. Might not be needed to be understood in depth, but add to the confusion when they are.
Developer Tools	IDEs, CLIs, MCPs, skills, extensions and plugins are the interfaces through which developers interact with AI. Each with different capabilities and different assumptions about workflow. The pace of development is crazy, and even for a seasoned specialist often intolerable.
Context	Size limits, engineering, rot, compacting, transparency issues are the single biggest quality determinant in AI-assisted development. Unfortunately it's hard to understand and the tools are pretty bad at visualizing it. The importance of context engineering and <i>developing necessary developer 'context awareness'</i> is a thing that needs to be understood and mastered to get good results.
Documentation	Custom instructions, agent definitions, skills, architecture decision records, UX specifications, stack descriptions, task management artifacts constitute the written layer which determines whether the code aligns with your project's reality or drifts into generic patterns.

The role disruption

In addition to learning a lot of new concepts, we might need to rethink our roles in our field.

I've summarized my thoughts and visions on how the established roles and well understood job descriptions might change (or 'get realigned' in corporate) in this new world, and what new skills and focus areas they might need to develop along the way in the table below.

Role	Traditional Focus	Now Facing	Impact
Product Owner	Requirements clarity, prioritization	How specific must specs be if AI writes code? Can AI help refine requirements?	Must learn to write specification-like requirements
Architect	System design, technology choices	Can AI design systems? How do you architect for AI generation?	Must understand code generation patterns and searchability
Designer	UX/UI design, user workflows	Do UI designs become prompts for AI? How detailed must they be?	Must understand constraint communication
Team Lead	Velocity, scope, team cohesion	Uneven expertise across AI-capable tools	Must actively manage skill gaps and tool adoption
Developer	Implementation, problem-solving	When does AI replace judgment? What's left to learn?	Must develop new skills: spec verification, prompt design, governance
Tester	QA, regression, edge cases	Can AI find its own bugs? What does testing look like at scale?	Must understand AI-generation failure modes
SRE/DevOps	Infrastructure, deployment, reliability	How do you monitor AI-generated code in production?	Must develop observability for probabilistic systems

So if anything above is actually valid, we'll shift from *producing* to *governing*, from *doing* to *validating*. We'll explore what this means for each role in depth in [Chapter 6](#), but the pattern is already visible: every role needs to develop a new relationship with verification, specification, and trust. I'll admit: most of this is still speculation. Nobody has properly battle-tested it, at least in the long term.



Let's not throw the good things about Agile away when jumping on **Agentic Spec Driven Development** bandwagon. Let's at least keep the **Self Governing Teams** and **Continuous Feedback Loops**, and this time (let team really) live by them. People are smart and given trust and responsibility, they will figure out how to make it work. Later we'll also discover that the new set of tools and processes might give us entirely new roles to fill.

The team challenge: How to work together with AI

Essentially, developers and teams face the same question: how do you let AI agents 'do the work' while maintaining the accountability, traceability, and quality that professional delivery demands?

All of the above is hard enough even without AI, even when you have clear roles, a mature toolset, and processes. Be prepared that in the current wave you will get neither of these off the shelf.

- i** Team leads, lead devs, project managers et al need to figure out:
 - How do you **plan, structure** and **organize teamwork** now that the coding part is increasingly automatic, tools are immature and constantly changing?
 - What kind of **new roles, feedback loops** and **processes** do I need to fill, onboard, develop, execute and keep track of?
 - How to get the most out of this technology and remain sane, on budget and on schedule?

This book aims to give you some answers on these concerns.

Planning and coordinating work

There are several approaches to organize software development into manageable chunks. They range from very lightweight and flexible (vibe coding) to very structured and rigid (waterfall).

Each has different implications for how AI can be integrated and governed.

Some approaches strive to produce useful results early on, some only target go-live later in the process. The most lightweight is no plan at all: prototyping. It has been recently renamed to "vibe coding."

Here's a summary of the approaches and their implications for AI-assisted development (and perhaps in any software development context):

Approach	Chunk Size	Outcome Delivery	Validation Timing	AI Suitability
Vibe Coding	Arbitrary; follows intuition	Continuous experimentation	After the fact	Excellent for exploration; poor for production
Iterative/Agile	Small stories/tasks	Early and frequent releases	Each iteration	Good with proper specs and tests
Phase-Gated	Medium features/milestones	Per phase completion	End of each phase	Requires thorough phase specifications
Waterfall/Big-Bang	Large; entire system	Single go-live	End of project	Poor; late feedback

The choice of approach (which typically has properties from several of the categories above) dramatically affects how AI assistance can be governed.

Vibe coding maximizes AI's creative potential but sacrifices traceability. A waterfallish approach demands specifications AI can follow but provides feedback quite late. Iterative approaches often offer the best balance: structured enough for governance, flexible enough for AI's strengths.

Recent critiques of AI-powered software engineering, and even SDD, miss the point though. Doing proper planning up front does not mean that the entire process needs to be the old 3-year waterfall. Even vibe coding, as implausible as it might sometimes still be, can be a useful tool for instance to prototype a new user experience feature.

For me, the spec-driven approach feels quite natural. Those who entered the business after the Agile Revolution will have a harder time adjusting.

It should be unsurprising that you get better results if you have a clear agreement before the work. We just stopped doing that for a decade or two in IT. What makes this time a bit different is that now the coder might actually read the specs.

Context engineering to the rescue?

Now on to the next big word in AI-powered software development: context engineering.

What it really is: when you call an LLM to work for you, you pass in a long text called a prompt. Perhaps you attach some files, and the AI does some lookups, and responds. That's the **Context**. Think of it like a transcript of your chat.

As the name suggests, **context engineering** is the collection of techniques related to what gets in that context. As in discussions, you should be very careful what to say to keep it on track and on subject, and include the relevant details for the other party to do its job, decision, or whatever.

All this is kind of quite complicated with AI.

When we strip the agentic software factory into bare essentials, it's worthwhile to note that essentially we're still crafting prompts and getting some response. All an 'agent' is is a multi-step prompt parsing, with some added capabilities via tools. The fundamental model is deceptively simple:

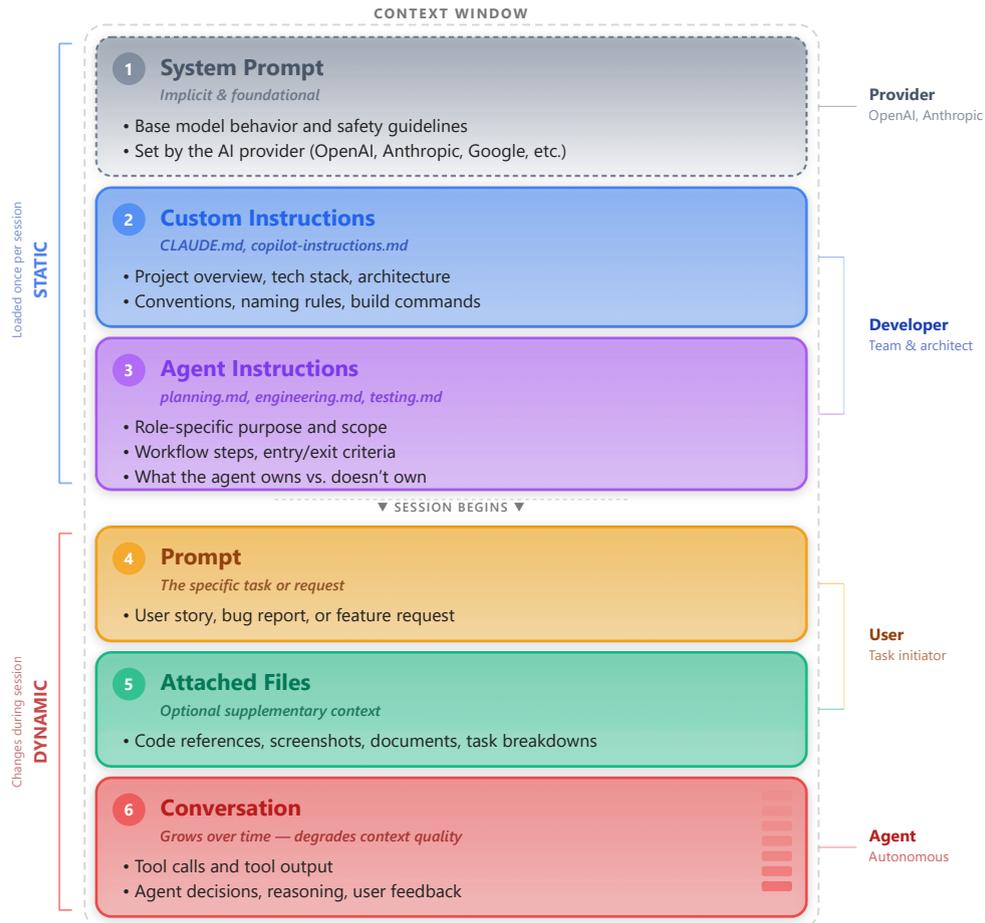
input → **f(x)** → **output**

The **input** is the context: your prompt, system instructions, conversation history, tool results, file contents, i.e. everything the model sees. The **f(x)** is the transformer: a statistical function that predicts the most likely continuation. The **output** is the generated response. Even though the LLMs don't have a concept of session, but everything is a 'post back' as in the old days of web development, the context is essentially the session state. Kind of 'transient component' if you will.

! The quality of your output is bounded by the quality of your input. No amount of model capability compensates for a poorly constructed context.

Anatomy of an Agent Context

Six layers that shape every AI decision



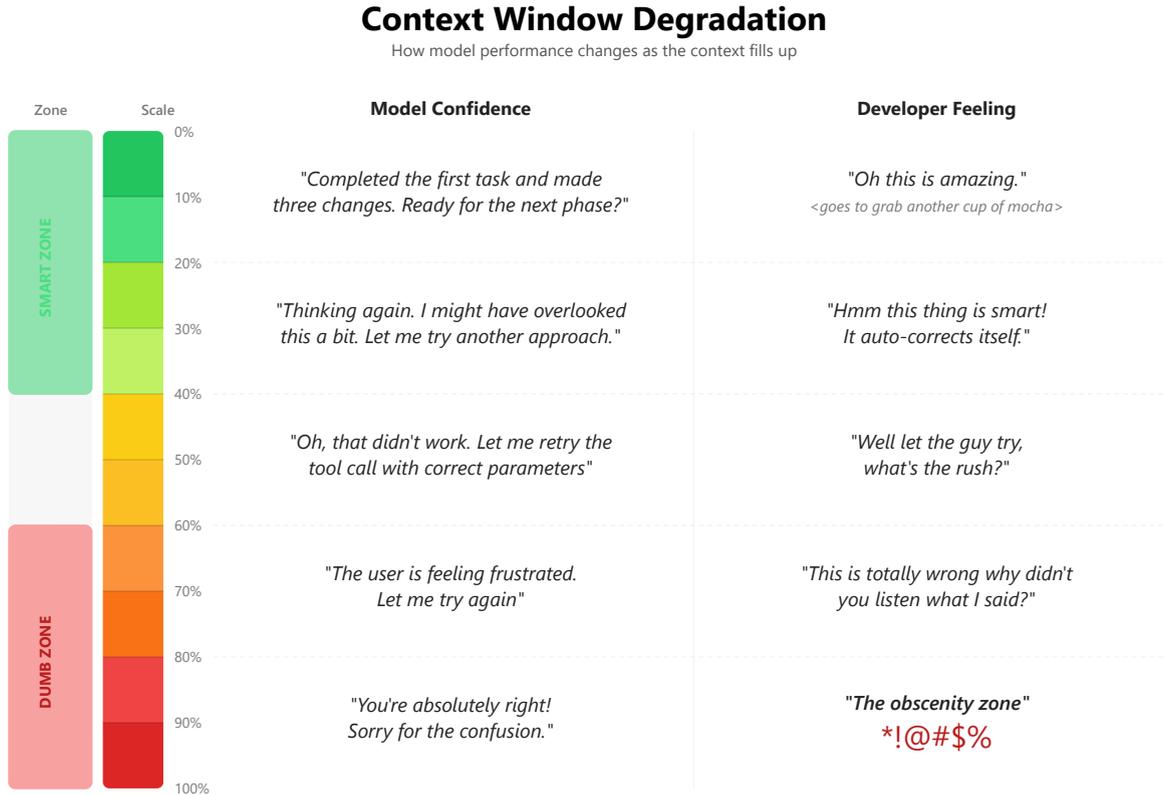
And here's where it gets worse.

Despite improvements in context window sizes, up from 4K tokens in early GPT models to 200K+ in current ones, limits still exist, and bigger isn't always better. Context can be *rotten*: filled with conflicting instructions, outdated information, irrelevant code, or redundant history. As the context grows, the model's ability to attend to the right information degrades. Quality decays not because the model got dumber, but because the signal-to-noise ratio in its input dropped.

Context engineering, then, is the discipline of crafting, curating, and maintaining the information that feeds AI systems. It spans all dimensions of software development from requirements to architecture to code to tests. Arguably, it's actually *the* reason behind most of the tips, techniques, and practices I'm discussing in this book.

You could think of context engineering as kind of a continuum from **prompt engineering**), but this time it's the agents/LLMs doing the prompting, not you. Put simply, it's about ensuring AI has the right information at the right time, and nothing more.

I've illustrated the challenge with context in my Developer Feeling Meter (tm) below. I'm pretty sure it's rather familiar for those who've worked with LLMs for a while.



✔ Be smart and stay in the smart zone. Say /clear or press that + for a new session when the things start to get worse. Don't try to force it through with more tokens or more iterations. It's not going to work.

Too many tools, not enough time

According to the JetBrains 2025 Developer Ecosystem Survey, 85% of developers now use AI-assisted tools. But which tools and to what extent? There is a strong lineup of tools for you to pick: GitHub Copilot, Cursor, Windsurf, Claude Code, Antigravity, Codex CLI, Opencode, each with slightly different strengths, context management approaches, pricing models, and feature sets.

➤ — JetBrains. (2025). State of Developer Ecosystem Report 2025. <https://devecosystem-2025.jetbrains.com/>

Another very recent study conducted by the Practical Engineer blog discovered that 95% of respondents used AI tools for software development at least weekly.

https://open.substack.com/pub/pragmaticengineer/p/ai-tooling-2026?utm_campaign=post&utm_medium=email

But regardless of which specific tools you pick, the underlying architecture is converging. In [Chapter 18](#), we'll map how users, models, prompts, agents, instructions, files, tools, skills, and integration protocols like MCP (Model Context Protocol) and LSP (Language Server Protocol) fit together as a layered system.

As always, you need some practice around choice of tools: how to assess new tools without disrupting ongoing work, how to standardize without stifling individual productivity (remember emacs vs vi), how to share learnings across the organization.

This pace and volume of new development gadgets isn't a reason to avoid AI. I think but it's a reason to invest in the *governance layer* around them, which is exactly what this book proposes, and judging by the fierce competition to reach feature parity, remaining as Agent Harness agnostic as possible might be a good idea to wait until the smoke clears and the victors emerge.

And in the end, we'll end up something from Microsoft, Google and Amazon, right?

I am aware that the odds are that many things I assert in this book are going to be outdated in a year or two. Yes, it's possible that we have that Automated Agentic Software Factory doing all and anything we used to do for a living this just around the corner. You know, coming out of the beta right next quarter.

A poster on a Pub's wall: 'Free beer tomorrow'

Why this matters

There are two big risks that I see resulting from ingesting too much new, often confusing and not that well defined, vague concepts in a short time.

Taking the easy path. When overwhelmed by the volume of new concepts and tools, teams default to the path of least resistance: let AI generate code, skip the review ("it looks fine"), merge it, move on. The result is a codebase that nobody truly understands, with subtle bugs that surface weeks later and architectural drift that compounds over time.

Dismissing the whole thing. After a bad first experience like a hallucinated API, a generated test that tests nothing, an agent that rewrote the wrong file, teams conclude that AI coding tools "aren't

ready" and go back to doing everything manually. They miss this impressive possibility to up their game because the first impression was poor and there was no framework for understanding why.

Both of these responses are understandable.

In the rest of this book, I'll introduce some tools and observations to help you and your team navigate between these extremes.

3

Why This Is Harder Than It Looks

What the Evidence Actually Shows

The narrative around AI coding tools is relentlessly optimistic. Wild claims such as one-shotting a C compiler (at \$20,000 compute cost) emerge almost daily, but results rarely withstand closer scrutiny. Much of it is tech bro hype, and the sources are not exactly academic.

I think overselling the AI wave is counterproductive and hinders the effective rollout of these capabilities due to often totally warranted skepticism. The ground truth is more complicated.

! **Feeling productive and being productive are different things. The evidence shows a consistent gap between what developers *perceive* and what the data *measures*.**

The research cited in this chapter reflects data available through mid-2025. This field moves fast. The specific numbers will age; check the sources for updates. The patterns they reveal, however, tend to be more durable than the data points.

The perception gap

The METR study was a randomized controlled trial tracking 16 experienced open-source developers across 246 real-world tasks. These were not AI novices. Participants were selected for AI tool experience and used their own preferred setups (primarily Cursor Pro with Claude 3.5/3.7 Sonnet). The finding: developers using AI tools took 19% *longer* to complete their tasks than when working without AI. Before the study, they predicted AI would make them 24% faster. After experiencing the slowdown, they *still believed* AI had sped them up by 20%.

Sixteen developers is a small sample, and the specific 19% figure could be noise. But the *perception gap*, a 39-percentage-point chasm between what developers believe and what the clock shows, is large enough to be directionally meaningful. And it doesn't stand alone.

Google's 2024 DORA report, surveying over 39,000 professionals, found that every 25% increase in AI adoption corresponded to a 1.5% dip in delivery throughput and a 7.2% increase in delivery

instability. You guessed right, most participants reported *feeling* more productive nevertheless.

What DORA actually measures

Worth pausing on what DORA is, because it's probably the most widely accepted baseline for measuring software delivery performance. And not very often used. I haven't personally ever measured or seen these in the wild, but I've heard DevOps advocates preach about them in the past. That does not mean they're not real or valuable metrics, and I think the AI software factory community, of which I've been a part for a while should develop these further.

The research program, originating from the *Accelerate* book by Forsgren, Humble, and Kim, defined four key metrics:

Metric	What It Measures
Deployment frequency	How often code reaches production
Lead time for changes	Time from commit to production
Time to restore service	How quickly you recover from failures
Change failure rate	What percentage of deployments cause incidents

These four metrics consistently distinguish high-performing teams from low-performing ones, regardless of technology stack or industry. They've been validated across over a decade of State of DevOps surveys.

What makes the 2024 DORA findings on AI adoption so interesting is that they use this same established framework. The report doesn't measure "lines produced" or "developer satisfaction", it measures the delivery outcomes that the industry has broadly agreed matter. And against those metrics, increased AI adoption correlated with *worse* outcomes. Even as teams reported feeling more productive.

The **2025 State of AI-Assisted Software Development** report from Google DORA tells a more nuanced sequel. Surveying roughly 5,000 professionals, it found 90% now use AI at work, up 14 percentage points from the year before. Here's where it gets interesting: the throughput penalty from 2024 has flipped. AI adoption now correlates with *higher* delivery throughput, higher code quality, and higher individual effectiveness. But, and this is a significant but, delivery instability still goes up. Friction stays flat. Burnout stays flat. The report's framing is that AI is an amplifier, it magnifies the strengths of well-run teams and the dysfunctions of struggling ones. Organizations with strong foundational practices, things like clear AI policies, healthy data ecosystems, and quality internal platforms, see the benefits multiply. Those without them just get more of whatever problems they already had. That's basically the thesis of this book in a single research finding.

So the tools are being adopted at speed, the productivity *feeling* is real, but the delivery metrics tell a different story. DORA gives us the vocabulary to have that conversation honestly.

The screen recording data from METR reveals where the time goes: developers spent roughly 9% of total task time just reviewing and cleaning up AI-generated code. They accepted fewer than 44% of suggestions, and even when they accepted code, 56% reported making major changes afterward. The productivity gains from generation are consumed by the overhead of validation, and the more experienced the developer, the worse the tradeoff, because experienced developers were already fast.

i These studies all have limitations, and they disagree on specifics. METR is rigorous (randomized, screen-recorded, real-world tasks) but tiny. DORA has massive sample sizes but relies on self-report. Stack Overflow captures sentiment, not delivery outcomes. No single study settles the question. What matters is the convergent pattern across all of them: developers consistently *feel* faster than the delivery metrics suggest they are. When someone cites "10x productivity," ask what the methodology was, and whether it measured output or feeling.

Trust erosion

Stack Overflow's 2025 survey shows only 29% of developers trust AI tool outputs, down from 40% a year earlier. 66% report spending more time fixing "almost-right" AI code than they save. 75% still prefer asking another person when unsure. Yet 46% of developers actively distrust AI-generated output while usage keeps climbing.

The tools feel good to use. They reduce cognitive load on individual keystrokes. But the aggregate effect on delivery is not what the marketing suggests.

Code quality and technical debt

GitClear's analysis of 211 million lines of code found "code churn" (code rewritten or deleted within two weeks) has doubled since 2021. Code duplication linked to AI is up 4x (8x between 2020 and 2024). Copy/paste is now more common than code reuse.

These metrics deserve some nuance though. Code reuse is not always the virtue we like to think it is, and AI-generated code tends to be structurally different from what humans write. It's simpler, more verbose, heavier on comments, and less likely to reach for abstract OO patterns or clever indirection. That means more lines and more apparent duplication, but not necessarily worse code. High churn could mean fragile output, or it could mean teams are now willing to rewrite and refactor at a scale that was previously too expensive to attempt. The jury is still out on what these shifts mean long-term.

What is harder to dismiss is the issue rate. CodeRabbit's analysis of 470 pull requests found that AI-generated code creates 1.7x more issues than code people write in open-source PRs. Fu et al. (2025) analyzed 733 real Copilot-generated snippets from GitHub projects and found security weaknesses in 29.5% of Python and 24.2% of JavaScript code, spanning 43 CWE categories including code injection and cross-site scripting.



The code quality evidence is the most concerning finding in this chapter. Speed gains mean nothing if you're accumulating technical debt faster than you can pay it down. This is why governance matters. It's not about slowing down, it's about not creating problems faster than you solve them.

The organizational reality

The vibe coding movement — Andrej Karpathy's term for "letting AI write your code while you embrace the vibes and forget that the code even exists" — has produced a flood of prototypes, MVPs, and demos. It has also produced what multiple analysts now call a coming technical debt tsunami.

Inconsistent coding patterns emerge because AI generates solutions based on different prompts without a unified architectural vision. Documentation becomes sparse because the focus shifts to prompt engineering. Security vulnerabilities appear because models lack awareness of security implications.

Stack Overflow captures the team dimension precisely: AI's most recognized impact is on personal efficiency. Only 17% of agent users report improved team collaboration. That was the lowest-rated impact by a wide margin. This suggests that AI helps individuals, but it doesn't improve how teams deliver together as much.

Adoption and economics

- The AI code tools market reached \$7.7 billion in 2025
- GitHub Copilot achieves a 46% code completion rate, but only ~30% is accepted by developers
- JetBrains found 85% of developers regularly use AI tools, 62% rely on at least one AI coding assistant, and 15% have not adopted AI tools at all
- Y Combinator's Winter 2025 batch: 25% of startups running on 95%+ AI-generated codebases

This finding (individual productivity up, team productivity flat) is the gap this book exists to address. Parts II and III are entirely about turning individual AI gains into organizational capability.

- Stanford: employment among software developers aged 22–25 fell nearly 20% between 2022 and 2025

The regulatory dimension

The EU AI Act entered into force in August 2024 and becomes broadly enforceable in August 2026. High-risk rules for embedded products follow in August 2027. Fines reach up to €35M or 7% of global turnover for prohibited practices.

The EU Cyber Resilience Act requires secure-by-design development, mandatory risk assessments, and ongoing security updates for 5+ years.

Stanford HAI's 2025 AI Index reports that AI-related security and privacy incidents rose 56.4% from 2023 to 2024.

Teams using AI agents to generate code need to answer: who is liable when an agent introduces a vulnerability? How do you demonstrate that your development process meets regulatory standards?



The regulatory question is not hypothetical. By August 2026, organizations deploying AI in high-risk contexts will need to demonstrate compliance. A governed pipeline with traceability (the subject of [Part III](#)) is one defensible answer.

What the evidence tells us

The evidence cited above doesn't suggest we should stop using AI. It does suggest we're still in the early days of figuring out how to adopt this new way of working. The perception gap, the trust erosion, the code quality concerns, the organizational blind spot. These are symptoms of a technology adopted faster than the practices around it.

One reservation I want to flag honestly: nearly all available studies measure short-term outcomes. Task completion time, sprint velocity, code produced per week. Nobody has yet tracked a governed AI pipeline, or an ungoverned one, over two or three years in production. The initial productivity story might look very different once maintenance costs, accumulated technical debt, and team turnover enter the picture. We don't know yet. That uncertainty cuts both ways, it could be better or worse than the early signals suggest.

Which brings us back to the basic intent of this book: what engineering practices make probabilistic outputs reliable enough for production?

References and further reading

➤ <https://metr.org/blog/2025-07-10-early-2025-ai-experienced-os-dev-study/> — METR (2025). Early 2025 AI-Experienced Open-Source Developer Study.

➤ <https://cloud.google.com/devops/state-of-devops> — Google DORA (2024). Accelerate State of DevOps Report.

➤ <https://survey.stackoverflow.co/2025/> — Stack Overflow (2025). Developer Survey 2025.

➤ https://www.gitclear.com/coding_on_copilot_data_shows_ais_downward_pressure_on_code_quality — GitClear (2024). Coding on Copilot: AI's Downward Pressure on Code Quality.

➤ <https://www.qodo.ai/reports/state-of-ai-code-quality/> — Qodo (2025). State of AI Code Quality Report.

➤ <https://www.coderabbit.ai/blog/state-of-ai-vs-human-code-generation-report> — CodeRabbit (2025). State of AI vs Human Code Generation Report.

➤ [Forsgren, Humble, Kim](#) — Accelerate: The Science of Lean Software and DevOps (2018). The foundational DORA metrics research.

➤ <https://arxiv.org/abs/2310.02059> — Fu et al. (2025). Security Weaknesses of Copilot-Generated Code in GitHub Projects. ACM TOSEM.

➤ <https://dora.dev/research/2025/ai-assisted-development/> — Google DORA (2025). State of AI-Assisted Software Development.

➤ <https://www.jetbrains.com/lp/devecosystem-2025/> — JetBrains (2025). State of Developer Ecosystem Survey.

➤ <https://hai.stanford.edu/ai-index> — Stanford HAI (2025). AI Index Report.

4

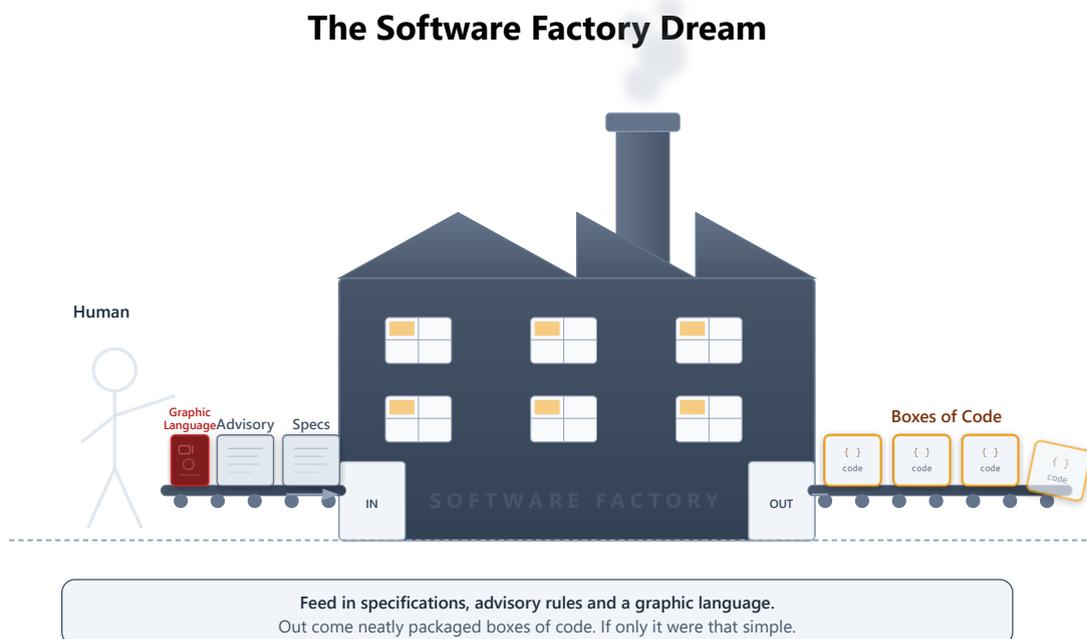
Everyone Wants a Software Factory

Power tools, not assembly lines

Everyone wants a software factory

The tools we covered in the previous chapters (CLIs, IDE assistants, and prompt-to-app builders) are coding accelerators. They are designed to help individuals work faster. But the topic of this book goes beyond individual tooling. It's about how teams share and develop the process and the AI framework that brings it all together. Essentially people want a **software factory**.

Of course this is compelling. Just describe what you want, and a pipeline of AI agents delivers working, tested, deployed software. A software factory. Simple, right?



To put it mildly, and as someone who has been working with (real) manufacturing for best part of their career by writing real software for it, I found the metaphor 'software factories' quite misleading. It simplifies the art of software engineering to a mechanistic process of assembly line production,

which is not how software development works or what is the outcome of it. The closest thing to 'factory' and 'creating software' in the industrial economics terminology I could think of could be 'tailored mass production'. If you leave the mass out of it.

Something from the construction might be closer. How I see it is that writing software is **akin to first designing and then building one-off buildings or complexes**. Not spewing out standard blue LIDL stores.

Yeah, we might have some templates, regulations and standards (perhaps quite a few of them) to follow. Like software, most buildings have a roof, walls, windows and doors. But the details are different every time! A site for storing nuclear materials, which arrive on railway and you need to build a hotel for IAEA inspectors to stay in (think like, SAP), are quite different things, from let's say a shopping mall (like, your custom CRM) or a suburban single family house (your personal webpage with javascript animation on startup).

Let's not dismiss the factory metaphor entirely, though. The construction analogy is closer to how software actually works: one-off designs, unique constraints, skilled judgment at every turn. But "software factory" captures something real about the ambition: repeatable pipelines, automated quality checks, consistent output. The industry has settled on the term, and so will this book. Just keep in mind that the "factory" we're talking about looks more like a custom design shop with very good power tools than an assembly line stamping out identical widgets.

@ Building the roguelike with GSD was exactly this: describe the feature, let the research agent dig into Lovecraftian lore and roguelike conventions, review the plan, then let the coding agent execute. A one-person software factory.

What's being built

Let's first look at what kind of factories already exist.

AWS launched Kiro as a spec-driven IDE in mid-2025, but by year's end it had evolved into something more ambitious. The Kiro autonomous agent can run for days, working across multiple repositories, opening pull requests with detailed explanations, and learning from code review feedback. It runs up to ten tasks concurrently in isolated sandbox environments. Spec-driven methodology (requirements in EARS (Easy Approach to Requirements Syntax) notation, design documents and test plans) is baked into the workflow, not optional.

Microsoft published a detailed reference architecture for an end-to-end agentic SDLC using Azure and GitHub: Spec Kit handles the planning phase, GitHub Copilot agents handle implementation, and GitHub Actions orchestrate CI/CD. Separately, Microsoft's Data Science team proposed an architectural concept using specialized AI agents for each SDLC phase: requirement extraction,

system design, code generation, testing, deployment, monitoring. These would all be coordinated by a Core Orchestrator Agent built on Azure AI services.

EPAM built a custom agentic platform for PostNL that deployed over 20 types of AI agents supporting multiple teams across the full lifecycle: test case generation (80% time savings claimed), documentation (70–90% reduction in manual effort), code review, and deployment. This is one of the few documented enterprise case studies of an actual production deployment, not a concept or reference architecture. I don't have any data on reliability or performance, so I'd approach these claims with a healthy dose of skepticism though.

Cognition AI's Devin and the open-source **OpenHands** (64k+ GitHub stars) take a different approach entirely: a single broad-capability agent that plans, writes code, browses documentation, debugs, and deploys. No role-specific stages, no formal handoffs. Assign a GitHub issue, and the agent figures it out. OpenHands achieves 72% resolution rate on SWE-Bench Verified.

As important pieces of these larger factories, **Task organization and learning tools** such as **Spec Kit** (and its forks), **GSD** and **Beads** don't try to own the full lifecycle but add crucial structure around existing coding agents. It's actually amazing that the most widely used AI developer tools don't have any proper tools to manage projects, tasks or progress. (No, nobody wants to use JIRA for that in the future. Never wanted.)

For instance, as my weapon of choice to guide my way through the Lovecraftian adventures in my roguelike, GSD helped tremendously in breaking work into phases and tracking progress in planning artifacts. In my day job I've built something like that (but far inferior), too. GSD is based on a (kind of) agent workflow to use specialized workers for specific tasks. What I found to be particularly well thought out in GSD is the interview process, where things are constantly refined and the user is asked for input before jumping into the coding agent. Beads provides persistent memory so agents can wake up, ask "what's next?", and resume work across sessions.

i With the possible exception of Microsoft's Azure- and GitHub-integrated reference architecture, most of these efforts stop at the PR. CI/CD, monitoring, and user feedback loops are left as an exercise to the reader. More about these loops later in this book. I think this is a interesting opportunity. Combining data from infrastructure, usage patterns and user feedback would provide trace data to analyze both the qualitative and quantitative performance of the produced solution, and feed good insights back into the next iteration.

References

➔ [EPAM PostNL Case Study](#)

↗ [Devin by Cognition AI](#)

↗ [Spec Kit](#)

↗ [GSD](#)

↗ [Beads](#)

The **compound probability problem** I've introduced applies directly to each one of these solutions I listed and probably to all the GenAI-based ones I didn't mention.

Some may have some capabilities for self-correcting and so forth, but are essentially at the mercy of the persons using them and verifying the product really did what it was supposed to do. So any "92% accurate" agents or "70%" SWE bench models perform *way worse in real*, parallel multi-agent setups, outside their training material and with the noisy input and lackluster QC department. This should come as no surprise to anyone.

This may be partially addressed by manual intervention and iterative fixing. Yes, there are times when you launch a lot of agents with a large task list and let them self-correct and run for days (e.g. 'ralph looping'). Using this kind of approach, you should introduce reinforcement-learning style feedback, be token-efficient, and have an orchestrator that stops when things go sideways. Let's call that the circuit breaker. Certain tooling for this kind of approach is emerging, but it's still early days.

*Without good tooling, stop points, self-correcting features and detailed design, this approach is essentially **brute forcing** rather than something intelligent.*

I'm not sure any of the major players have good answers to this yet. Spec-driven methodology constrains each agent's scope, reducing the probability of drift.

In all honesty I haven't tried them all and I'm painfully aware that I might be outrun by my own competitors.

There may be technical guardrails like token budgets, timeouts or hooks to prevent runaway agents. But the "iterate until tests pass" approach is not a solution to the compound probability problem. It adds more iterations and can actually make things worse if the underlying reliability isn't high enough. And it is, by nature, not.

One thing constantly overlooked and partially solvable via smart agent design is related to testing. With vague tests, overenthusiastic agents, your 231 E2E tests with 100% are just code fitted to the code, not the real needs. (And often break when somebody as much as looks at them.) Without careful guardrails, like isolated testing agents that cannot read the implementation code, or do that after the fact, agents will eventually try to modify the tests to fit the code. Obviously, this defeats the purpose of testing entirely but is what is happening unless you pay attention.

Whether full-lifecycle agent orchestration can work *reliably* without enforced governance is the question the industry is running an experiment on right now. The rest of this book is about finding an answer to that question.

We'll return to these frameworks with a detailed tool-by-tool analysis later in this book.

5

The Compound Probability Problem

Why AI coding tools are fundamentally probabilistic

Inner workings of LLMs and the probabilistic nature of the technology

An LLM produces the *most likely next token* given its context when producing your output, token by token. On the other hand, software it produces must be *deterministic* which means the same input must produce the same output, every time.

Below is a simplified illustration of the problem. It does not aim to completely capture the complexity of the issue but to give a visual representation of the core tension. In the end, after feeding our prompt and all the instructions, the model produces a probability distribution over all possible outputs. The more specific and constrained the prompt and context are, the narrower the distribution becomes, and the higher the probability of getting what you wanted. Or at least that's the goal.

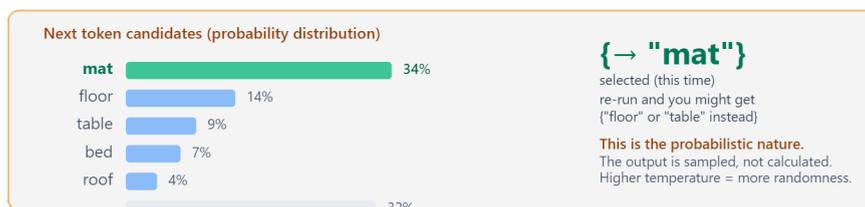
How an LLM Generates Text

From prompt to next-token prediction

"The cat sat on the ___"



What the model sees after "The cat sat on the":



Every output token is a probabilistic choice. Over hundreds of tokens, these choices compound — that's the challenge.

The 'thinking' happens also on a higher layer, in a way, as kind of internal back-and-forths of the model before the output is produced. As those who have used modern LLM-based software engineering tools know, i.e. there's much more in play than I included in the image, but the fundamental thing is the same: when spitting out the next instruction to your code, the model picks it from a list of candidates, and the more you narrow down the list, the better your chances are.

One of the approaches to make models better has been to increase the size of the model and get more training data. Up to this point, *this has indeed improved the quality of the output*. Also the size of the input, i.e. the context, has increased dramatically, so in theory, you are able to solve more complex problems in a single run, and having that in the range of 100k tokens is a big deal and a necessity to solve the more obvious, 'smart' moves. There are signs, however, that this approach is nearing its limits, and that we might need to start thinking about other approaches. Throwing more GPU/TPU cycles, 'ralph looping' or 'making parallel runs' on the problem is probably not the best way to go about it either.

The infinite monkey theorem postulates that a monkey hitting keys at random on a typewriter for an infinite amount of time will almost surely type any given text, including the complete works of Shakespeare. Let's not try this here.

Finally the training data problem. "The internet" has already been ingested, and the models have been trained on it. What happens to the quality of it when the training materials become more and more generated by the models themselves? From signal processing perspective, I'd expect the quality to degrade over time.

Keeping up with the research side on the LLMs would be a full-time job. Knowing the basics on how the technology works is still important, and in the core of my engineer's mindset: I could not imagine writing programs without knowing the basics on how a computer operates; CPU, memory, storage and so forth. For the Agentic/GenAI software development, similar basic building blocks are the transformer architecture, the attention mechanism, the training process: how your prompt (+context) is transformed into an output.

You'll understand how this all works (to certain extent at least) by just looking at the logs. The IDEs and assistants go to great lengths not to show them to you, but they're there for you to study and learn. What's the FULL context? What tools were really called and by what parameters? How many tokens were used and how long did it take? What was the exact output? These are the things you need to understand to be able to work with the technology in a more informed way.



TL;DR In GitHub copilot, it's "Chat Debug View". and in Claude Code it's in the jsonl files per session, and there are plugins around to view them nicely.

[↗ Lost in the Middle: How Language Models Use Long Contexts](#)

So when you give AI a precise specification, carefully curated context, and well-defined constraints, the odds of one-shotting your task are often pretty good. Give it a vague prompt and a million-line codebase with no documentation, and you get what probability gives you: plausible-looking code that may be subtly wrong in ways that take longer to debug than manual writing would have.

Frankly all this is not that much different if at all than working with people. They aren't perfect readers, have their biases and implicit assumptions as well, and also make errors. I've been entertaining myself with this: now we might finally have the near-perfect developer who at least tries to read the documents.

In summary, the AI framework and processes built around them need constant tuning, just as agile development methods always called for feedback loops. Agents wander? Check why. Wrong instructions being read or just ignored? Check the wordings and ordering, make sure there's no overlap or conflicting text. And so on. The inherently probabilistic nature of AI isn't a bug that will be fixed in the next model release or with increased GPU power, and you should think of them as computer programs invoking functions and sending messages. **It's the fundamental nature of the technology.**

The compound probability problem

Let's do a simple (and naive) calculation of what happens when you chain LLMs. When LLMs effectively produce each other's inputs, as is the case in agentic usage, the odds of success don't necessarily increase. At least not if the agents cannot validate their outputs or inputs and reject false ones. I'll refer to this as the **compound probability problem**. The more steps you have, the more chances for things to go wrong, and the less likely you are to get what you wanted.

For instance, let's consider a multi-agent pipeline with several steps: Backlog → Planning → Engineering → Testing → Review → PR. Each stage involves an AI agent making probabilistic decisions. Even if each stage is 95% accurate, the compound probability across four agent stages is only 81.5%:

$$0.95^4 = 81.5\%$$

If you've ever worked with reliability engineering, this is the same thing as Mean Time Between Failures (MTBF). Chain components in series and each one that can fail drags the overall reliability down. **Exponential decay**, the engineers call it.

In a way to make it worse, during a single 'step' an agent performs, there are several probabilistic decisions inside. Of course that didn't affect my 'master formula' above, but this illustrates the non-

constant nature of per-step accuracy. These kinds of intra-agent decisions include:

Tool selection	Which MCP server to call, which function to invoke, what parameters to pass. Each is a probabilistic choice. Wrong tool → wrong data → wrong code.
Document retrieval	RAG (Retrieval-Augmented Generation) retrieval is similarity-based (kNN or similar). You may get irrelevant docs, miss critical ones, or misinterpret what it reads. Stale docs poison the context.
Result interpretation	Even with the right tool and right docs, the agent must interpret results. A subtle misread of an API response or schema doc cascades into generated code.
Agent-to-agent handoff	Structured artifacts help, but the receiving agent still interprets them probabilistically. Any nuance lost in handoff accumulates across the chain.

So, if each of the agent stages involve six internal probabilistic decisions, a four-stage pipeline has 24 effective decision points:

$$0.95^{24} = 29.2\% \text{ (at 95\% per decision)}$$

$$0.97^{24} = 48.1\% \text{ (at 97\% per decision)}$$

These numbers are illustrative, not exact. The math assumes each decision is independent, which is a simplification. In practice, decisions are correlated: an early architectural mistake biases every subsequent step, and conversely, strong context can lift accuracy across the chain. But the basic point stands: the longer and more uncontrolled your pipeline, the worse it gets. So what can we do about it?

Input Quality	The more you can narrow down the space of plausible outputs, the higher the probability of getting what you wanted. Clear specs, curated context, and well-defined constraints are crucial.
Checkpoints/validation	Catch errors between steps before they propagate (like error detection codes)
Human-in-the-loop	Equivalent to a manual inspection step
Input size	Smaller tasks → fewer decisions → higher per-step accuracy
Shorter chains	Fewer steps = higher reliability (minimize series components)
Redundancy	Run the same call multiple times, vote/compare (like N-modular redundancy)
Self-correction loops	Retry with feedback, like automatic failover

All of these are going to be discussed in this book. Let's first add the most important one: make sure your input is correct.

Input quality

It's probably rather obvious that unless you are clear on what you want, you won't get it. But the probabilistic nature of the technology makes this even more critical. Things like wording, ordering, and formatting of the prompt and context can have a huge impact on the output. The more you can narrow down the space of plausible outputs, the higher the probability of getting what you wanted.

Roughly speaking, you can expect the following accuracy per decision based on the quality of your prompt and context:

- **Vague prompts:** 70–80% accuracy per decision (huge search space)
- **Good specifications:** 90–95% accuracy per decision (constrained space)
- **Excellent specifications + clear context:** 95–99% accuracy per decision (narrow solution corridor)

Fear not: this is not "big waterfall upfront design" I'm suggesting. It's *decomposition for probability management*. Breaking a large, ambiguous problem into small, specific tasks isn't waterfall, it's more like doing a detailed todo-list before you start coding.

A common straw man argument from AI skeptics is the notion that SDD is a return to the stone ages where everything needed to be written down up front. The key misunderstanding is the word "everything." Of course you need a solid roadmap, necessary architecture, and guidelines specified, but you can still work iteratively at the feature level and create feedback loops.

Think of it like this: in the old days you had the vague user story and some Figma designs often written by someone else ages ago. You took a quick look and figured you'd just start coding and see if you got it right. Then in the next sprint, you'd refine.

You can still do this, but you need to shift the discovery phase outside the coding, or major parts of it. What pages or controls are needed and for what exactly? What messages, tables, and APIs will you send, read, or call? How do you verify that the thing actually did what it was supposed to do?

Checkpoints and validation, human in the loop

Having gates between your agents is, as illustrated above, a good idea to prevent your premium tokens from going to waste. Putting gates between your steps doesn't just enforce process, it **breaks the probability chain**. A human checkpoint resets error accumulation.

So, instead of p^{24} (probability compounding across 24 decisions), you get gates resetting at checkpoints:

Scenario	Formula	Result
No gates, 24 decisions at 95%	0.95^{24}	29.2%
4 gates (6 decisions each at 95%)	0.95^{64} with corrections	~75% (plus human catch at each gate)
Meaningful gates with review	0.98^{64} with rework loops	~90%+ effective

Key is to catch drift at task boundaries. How it is done is perhaps something you'd imagine an UAT manual tester to detect if your beloved piece of software actually sucks at something or not; now we're gonna introduce that kind of sanity checks way earlier in the chain, shift left. How feasible it is to have a human in the loop depends, but if we rethink the role of this manual gating, it just might.

For example, instead of manually code-reviewing every PR, you could let that nice-talking agent to run static analysis, conformance, security and other checks and have you review them if necessary, granted you've specified quantitative or boolean criteria for your quality controller agent as triggers for manual intervention.

By applying checkpoints and validation, with optional manual review, we intervene early, evaluate what went wrong and try again with hopefully increased odds to nail it.



This is what the DevOps 'shift left' and 'continuous feedback' principles are all about. It's just that now we have to apply them to the AI-driven stages as well.

Input size

This one's straightforward but easy to ignore: **step size management**. The larger the task or step you ask an agent to do, the more decisions it has to make, and the more chances for things to go wrong. By breaking down your work into smaller, more specific tasks, you can increase the per-step accuracy and mitigate the compound probability problem.

Success Rate $\approx 1 / \text{Step Size}$

This principle applies to both architecture and feature. Smaller the step size for a feature, the higher the probability of success. More modular the architecture, the more constrained the solution space, and the higher the probability of getting what you wanted.

@ My Lovecraftian roguelike has about 200 different features. Each was specified, generated, and tested as a bounded task: a new character class, a map event, an inventory mechanic, all bound to wider themes and phased across the game. I think I'm at Stage 33 of Milestone 3 right now. I didn't call it "step size management," but that's what it was. Keeping each generation request small enough that compound probability never had room to multiply.

This is not a new idea, but it's more important than ever in the context of AI-driven development. Working like this was a good idea when developers still did the majority of the heavy lifting: think before you start throwing GPU cycles at it.

We learned this on a frontend project with complex screens. The planning stage had already decomposed features into small, atomic tasks, the kind of bounded work that should have been straightforward for the engineering agent. But the execution harness was handing the agent an entire feature's worth of tasks in a single session. So the agent tried to build everything in one context window: layout, state management, API integration, validation, edge cases, all at once.

What happened? The agent's context filled up, quality degraded toward the end of the session, and retries started piling up. The fix was almost too simple to admit: execute one atomic task per session, each starting from a fresh context. Same tasks, same plans, same specifications. Just fed to the agent one at a time instead of all at once.

Retries decreased significantly. Not because the tasks changed, but because the step size *as experienced by the agent* changed. We had planned small. We just weren't executing small.



Having small tasks in your plan is necessary but not sufficient. The execution model matters just as much: if your agent consumes all the tasks in a single context window, you've effectively created one large task with all the compound probability that implies. **Plan small, execute small, reset context between steps.**

Shorter chains

One of the core engineering principles for everything has been 'keep it simple'. Or, as Occam stated, "Entia non sunt multiplicanda praeter necessitatem, entities should not be multiplied beyond necessity".

The same applies to the compound probability problem. The more steps you have, the more chances for things to go wrong, and the less likely you are to get what you wanted unless you have good governance, checkpoints and circuit breaks. Narrow the tool scope, control the access to the codebase etc. Start with a minimal set like the standard 'Plan' and 'Do', and see if adding more agents and stages is really necessary for your use case.

Redundancy

Having dual power supplies and multiple hard drives in a server is a common practice to increase reliability. For the agents it's not that obvious, but you can run the same call multiple times and compare outputs. Better yet, use a **different model** for each run.

It's basically a voting system: run the same thing three times, compare, and pick the winner. Same idea as N-modular redundancy in hardware, where you have multiple components doing the same job and a vote decides the correct output. If the spread of answers is too wide, you might resort to having someone (in case they're not out to lunch or asleep already) to make the final call.

Self-correction loops

I'm going to discuss this topic in more detail later in the book, but the idea is probably clear: agents should learn from their mistakes, adjust the assumptions (tools, context, constraints) and try again. Most likely at first you'll be figuring out the necessary corrective actions manually, but eventually you can automate some of this process.

For instance, if an agent fails a test, it could automatically analyze the failure, identify the root cause (e.g., wrong API call, missing doc, misunderstood requirement), adjust its prompt or context accordingly, and retry the generation. Or on a higher level this could be figuring out why the damned thing created a new page instead of editing the existing one, i.e. why did you do this?

- ✓ Having this kind of self-healing learning capability to increase the reliability of your agentic system is a powerful way to mitigate the compound probability problem. But make no mistake, we're not quite there yet and much of the job description of the new engineering I'm going to suggest in this book would be about this part: how to make the agents learn from their mistakes and improve over time.

What remains unsolved

Even with excellent governance and all the various ways I described above the **compound probability problem** is never going to be fully solved.

The fundamental probabilistic nature of the technology means that there will always be a non-zero chance of failure at each step.

So looking at the crystal ball, we can expect the following as you head toward the promised land of AI-assisted development:

1. Model reliability continues to improve with better hardware and training, but remains probabilistic
2. The failure rate per step across a larger agentic system is uncharted territory and probably highly context-dependent
3. One-shot generation (entire systems from scratch) remains high-risk, expensive, and slow
4. Drift in base model behavior between releases can break carefully tuned pipelines.
5. AI-driven development remains genuinely hard to stabilize.

In the meantime, the best we can do is to understand the nature of the problem, apply the mitigation strategies, and design our systems with the compound probability problem in mind.

@ A hobby game with no users can afford a few probabilistic misfires. You just regenerate and move on. **Production systems can't : it's not an option to regenerate the whole damned thing every few weeks.** But the principle transfers directly: constrain the problem space per step, and the math works in your favor whether you're building a dungeon crawler or a payment system.

6

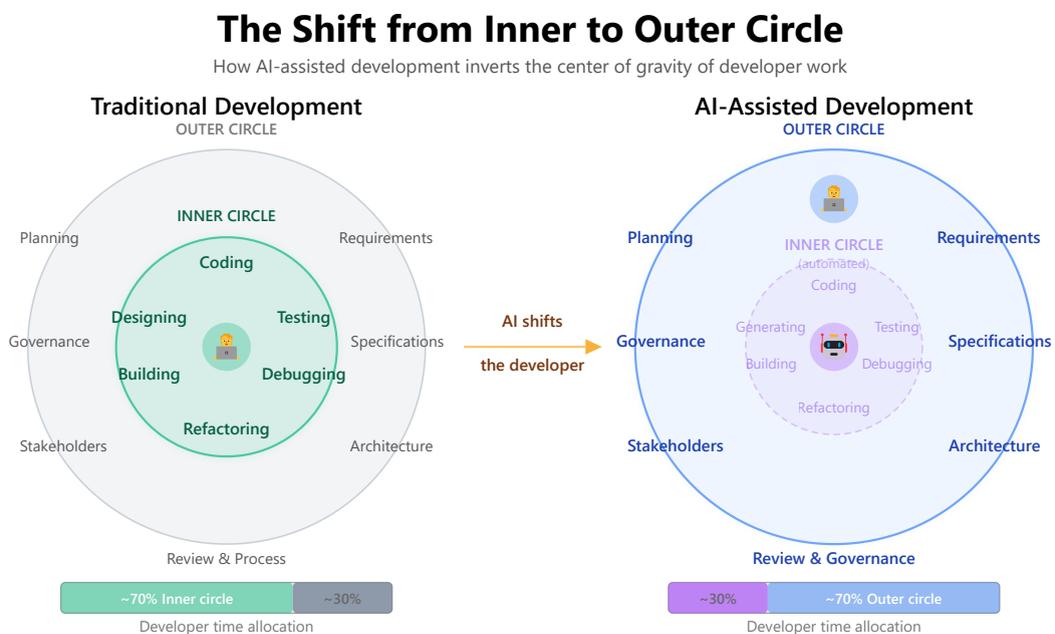
From Writing Code to Directing Agents

What actually changes when AI takes your (old) job

From writing code to directing agents

One way to look at how work has been traditionally organized in software development is to divide it into what involves code and what doesn't: the **inner** and **outer** circles. It's a useful metaphor in several ways: as developers and engineers we've been, for good and bad, quite obsessed with technology and code. The people and organizational aspects, or even the 'what' we're actually supposed to be doing, have (sadly) been a secondary concern. How I see it is that many issues with software projects have always been the disconnect between the two circles i.e. not technical, but human interaction problems. As engineers, we need to swallow the bitter pill and come to terms with the fact that our beloved inner circle or 'code' is no longer the "real work" or "real beef". The outer circle is.

Examples of tasks in these circles are of course not clear-cut, and the boundaries are blurry and connection points are numerous. Roughly speaking, you could think of it like this:



The inner circle	Code, test cases, pipelines, debugging, refactoring. Developers, testers, oddball architect, tech lead, cloud specialist, SRE.
The outer circle	Requirements, specifications, architecture design, aligning with stakeholders, reviews, workshops, UI/UX design, processes. Architects, tech leads, product owners, project managers, delivery leads, designers.

An alternative way to think about this is that the outer circle was what you did in meetings, or what the tech lead handled, or what happened before and after the "real work.". And the inner circle began when those guys split and you got back to coding.

@ Building this game, I never wrote a line of game logic. I wrote specifications, reviewed output, and made architectural decisions about module boundaries. I was already working in the "outer circle" this chapter describes. I just didn't have a name for it yet.

AI-assisted development adds a **new abstraction level** to our work. Getting serious about this means you need to actually make that leap. Now, even though AI might very well handle most of the typing, the inner circle doesn't disappear, but it shrinks, and more of your efforts will be in the outer circle: deciding what to build, specifying it precisely enough for an agent to build it correctly, reviewing whether the result meets the standard, and governing the process that connects these activities.

Naturally, this is a "shift left" in a much deeper sense than the DevOps community originally meant. Originally we shifted *testing* left, with the idea that writing tests earlier catches defects sooner (this of course still holds). Now the new shift left moves the *developer* left: from production to direction, from writing to specifying, from building to guiding.

! **The outer circle was never the part most developers trained for, or chose this career for, or built their identity around. It was the overhead. Now it's the job.**

Everything that follows in this chapter — what the daily work looks like, why senior engineers struggle, where junior developers go, how complacency creeps in, why the identity shift is hard — is a consequence of this inversion.

What actually changes Day-to-Day

Now that we're convinced this shift is inevitable, it's a good time to say what it really means. I'll throw in examples by role, and my assessment of what the future holds for us.



Expect cynical humor ahead. If you don't like it, skip to the next section.

Architects

	Before	After
Requirements	Analyze and specify the core building blocks, their interactions and the NFRs	Describe your architecture in a way that's accessible for AI. Same as before, but in a new way.
Alignment	Constantly ensure, as none of the above is connected to code unless you do it yourself	Refine these descriptions and test the alignment yourself. Develop automated ways to check,
Distance	Rarely code (I've been fortunate enough to almost always be hands-on big time)	Should check code, documents and their indexing, agents and their tooling frequently
Toolset	Draw diagrams and author documents and ADRs (Architecture Design Records)	Generate and refine drawings with AI, when needed
Where time is spent	Meetings	Meetings and throwing insults at computer when things don't work

Designers

	Before	After
Analysis	Carefully and iteratively refine the user experience, accessibility and workflows for coders to implement	Quickly develop a working prototype by ingesting a design system based
Distance	Depends, but usually never see the code, and not always really in the same pace or loop as the developers and testers	Need to be closer to development to align that e.g. the prototype you make is a viable starting point
Toolset	Various design tools (such as the one that begins with the letter F) and other visual tools.	Automated UI-focused AI platforms such as Lovable
Where time is spent	Meetings arguing with developers who didn't follow your designs	Trying to iteratively modify the solution made by Lovable and hurling obscenities at the computer

Developers

	Before	After
Analysis	Skip the specs, we'll figure this out as we go	Yeah I need to create a proper plan and a todo list before I can code
Distance	Well, you're as close as it gets, and the further away the architects and others are the better	The architect now makes commits to the codebase. Still undecided whether that's a good thing.
Toolset	IDEs, CLIs and git. Never touch the JIRA unless somebody forces you to.	IDEs and CLIs and git with an AI-wrapper on top of them
Where time is spent	Meetings that don't really concern you, and debugging funny issues. 10% real coding.	Debugging funny issues and babysitting AI agents going rogue

Testers

	Before	After
Analysis	Skip the specs, we'll figure this out as we go	Yeah I need to create a proper plan and a todo list before I can test
Distance	Waiting for developers to finish their work literally just behind their backs	I actually need to code or review the automatic tests myself
Toolset	Test frameworks, CI pipelines (in theory) and some mostly testing tools (reality)	.spec.ts and the agents that run them. Playwright trace files.
Where time is spent	Waiting for something to test or get fixed to be tested again	Looking at the trace files and trying to figure out what the heck happened and why the test failed

Cloud engineers, SREs

	Before	After
Analysis	It's the same as what is needed regardless of what is being built.	We still need the Kubernetes cluster
Distance	The further away the better, as long as you get the requirements right	You actually need to have your code in the same repository as the developers
Toolset	Terraform, Kubernetes, CI pipelines, monitoring tools	Agents writing Terraform and Kubernetes manifests, and monitoring tools that are more focused on the output of the agents and the quality of the code they produce

	Before	After
Where time is spent	Meetings arguing about how to do the infrastructure and firefighting when things break. Trying to figure out where and why the Terraform state file went out of sync with the actual infrastructure	Trying to figure out why the agent wrote a manifest that doesn't work and then trying to fix it yourself

Project managers

	Before	After
Analysis	A detailed 100-page document nobody, including you, ever reads, and a Gantt chart that is outdated the moment you create it	A short PowerPoint and bold promises "AI will do this in 1/10 of the time"
Distance	Weekly meeting with the team, and then you go back to your office and do the actual work of project management	Monthly meeting with the team
Toolset	Should've used: JIRA, Devops, Dashboard. Really used: Excel and e-mails	Should've used: Dashboards that track the progress of the agents and the quality of their output. Really used: Excel and e-mails
Where time is spent	Meetings and living in despair as the JIRA tickets never get updated anymore	Meetings and living in despair as the JIRA tickets never get updated

What's left for junior developers?

In traditional software engineering, junior developers learn by doing progressively harder work. They start with simple bug fixes, graduate to small features, and over several years build the pattern library and system intuition that makes a senior developer effective. The work is sometimes tedious, but the tedium is where learning happens.

AI-assisted development threatens this pipeline directly. If agents handle the pattern work that juniors learned on, how do they build judgment? If they never write a service adapter from scratch, how do they learn to recognize when an agent has written one incorrectly? If they never debug their own logic errors, how do they develop the diagnostic instincts that make review effective?

My point here is that in order to validate a plan or outcome, which is the supposed new role, you need to have an idea what it should look like, **know what the correct answer is**. You cannot tell what it is without experience. Failing to do so, as will be the case, we will have a problem. If you let a number of inexperienced juniors go AWOL with agents for a week, with all certainty they'll end up generating a month's worth of technical debt, dozens of hard-to-find subtle bugs, and a codebase that nobody understands. The senior engineers will be left with the task of cleaning up the mess, and

the juniors will have learned nothing but how to make a mess. Actually, this kind of roles ('openings for Senior Slop Fixers') seem to have already emerged.

i The Stanford statistic (employment among developers aged 22–25 fell nearly 20% between 2022 and 2025) suggests this isn't a hypothetical concern. The pipeline of *future* senior engineers is already being disrupted, and the consequences will take five to ten years to become visible.

➤ [Canaries in the Coal Mine? Six Facts about the Recent Employment Effects of Artificial Intelligence](#)

My take on the study is that if we fail to keep the juniors engaged and learning, the consequences will be severe.

Failing to address this, we will discover a generation gap: a cohort of experienced engineers approaching

retirement and a cohort of AI-dependent developers who never built foundational skills. All in all I think people should really learn to code, write SQL, debug and experiment before turning them loose with AI tools unsupervised. Otherwise they'll be just **rubber-stamping AI slop** and learning nothing.

In a way this isn't new. You know, the one guy who made the critical piece in your system didn't do any KT before retiring, and there's no documentation whatsoever?

I'm not making an argument against AI-assisted development here. It is an argument for deliberate investment in junior developer training programs that use governed AI development as a **teaching tool** and insisting on knowing the old way as well. In this program, or whatever, juniors should learn specification writing, review technique, and system design explicitly, rather than absorbing them implicitly through years of code production.

The automation complacency trap

! Aviation researchers identified "automation complacency" decades ago: when humans monitor automated systems, their vigilance degrades over time. Humans are reliably poor at sustained attention to systems that almost always work correctly.

This applies directly to governed AI development. A plan approval gate is only as good as the human's willingness to read the plan critically. A review gate is only as good as the human's attention to code they didn't write. If the AI's output is good 90% of the time, the human reviewer will learn, perhaps unconsciously, over weeks, to expect correctness and skim rather than scrutinize. The 10% of the time the output is subtly wrong is exactly when the human's attention is most likely to have lapsed.

And the output looks and feels good, down to the wording like 'Your code is now ready for production.'. It's entirely understandable to skip the hard part of verifying the work and just approve, as the literal quote from a generally used LLM already said we're done.

The literature suggests several mitigations that translate to software governance. Rotation of review responsibility prevents any single person from becoming complacent with a particular agent's output style. Varying the level of AI autonomy on purpose, like running agents with deliberate constraints that require more human input could keep reviewers engaged. Surprise audits of gate approvals, where a second reviewer evaluates whether the first reviewer caught issues that were present, keep the reviewers honest too. Perhaps this might feel like overkill, but it is something you'll need to address to mitigate this trap of trusting the automation too much.

None of these are magic bullets. Automation complacency is a deep feature of human cognition, not a process deficiency. Acknowledging the facts on the ground, rather than assuming that "mandatory human checkpoints" will automatically produce careful human review is the difference between governance and governance theatre.

The new role nobody planned for

This discussion about roles and competences reminds me of the advent of cloud computing. Not that long ago, developers (often with somewhat reluctant help from sysadmins) managed their own infrastructure. Provisioning servers, configuring networks, and maintaining deployments were part of their jobs. It was not uncommon for the same person who wrote the software to also install the server, set up the databases, and handle whatever else was required. Perhaps the sysadmins found their new home as **SREs** or **Cloud Engineers** who struggle to keep their Kubernetes clusters alive, but for the most part, the cloud abstracted away the infrastructure and made it a side responsibility.

As a sidenote, I've missed the dedicated DBAs from time to time. That's a kind of lost art.

This worked to some extent when the commercial cloud offering was simple, but it grew complex enough that it couldn't remain a side responsibility. You've heard it all: dozens of certifications, consoles, IaC formats, well-architected pillars, the N Rs of cloud transformation, and so forth. The "cloud engineer" emerged not because anyone planned the role, but because the work demanded it.

AI-assisted development requires something similar. Whether we call this new job description the **AI Agent Engineer**, or the **AI Governor**, or the **Agent Orchestrator** does not make much difference. The point is that someone needs to own the responsibility of maintaining the agentic development environment, and that responsibility is not optional. This role includes but is not limited to:



Maintain the **custom instructions** that guide every agent interaction.

Design and curate the **agent configurations**: which models for which tasks, what constraints, what context.

Keep the **documentation hierarchy** and all of the above current and well-indexed so agents get useful context rather than noise.

Keep up with the **tooling and model developments**

Actively look up new **features and capabilities** that could be useful for the project, and experiment with them to understand how they work and when they are useful.

And this is iterative and continuous. I think it's unreasonable to expect this to be everybody's job or a shared responsibility. It requires a level of attention and care that is hard to maintain when it's not owned by anyone. Whether this is a full-time job remains to be seen and is obviously dependent on the project.

In small teams, this easily falls on whoever cares most. Usually this falls to a senior developer who has developed intuitions about what makes agents effective, and perhaps was an early adopter of AI in the first place. But as teams grow, the "whoever cares most" approach breaks down the same way "whoever manages the servers" did. The work requires dedicated attention, systematic maintenance, and a skill set that combines understanding of the development process, agent capabilities, and the project's evolving needs.

Organizations that recognize this early will create the role deliberately. Some won't, and will discover it through slowly degrading output quality, stale documentation that agents can no longer usefully consume, and instructions that reflect how the project worked six months ago. The cost of not having someone own this responsibility is invisible until it isn't.

Remember we needed that one extra 'Slop Cleaner' in the team? Well here's another addition. The 5X speed might not come for free after all.

The identity shift might be the hardest part

Many software developers chose this career because they enjoy building things. Myself included. Also quite a few of us began programming long before entering the job market. The satisfaction of solving a puzzle, of seeing your code work, of creating something from nothing can be lost forever when delegating that part to AI.

And this shift is a matter of identity. Many take pride (often well deserved) in their craftsmanship of writing good code, clean architecture, or beautiful design, if it wasn't really you doing it. You know the scene: you meet a fellow developer and the first question is "what do you code in?", not "what do you govern in?".

Working with an AI agent is more like managing a team than writing code. Or sometimes like working as a **kindergarten teacher**. You set direction, review output, make judgment calls, and accept responsibility for results you didn't directly produce. Some developers find this deeply satisfying: the power you can unleash is intoxicating, and the problems are harder. You can build tools and make refactoring decisions you would have dismissed outright before as too laborious or risky. For others it feels like a **demotion**: instead of engineer or builder, you become some kind of "orchestrator" or "curator" or "governor." The work is more about people and process than about code, and that can be a hard shift to make.

Anecdotally, the long-timers seem to be the ones who get on with it more naturally. Perhaps they've already typed enough code for a lifetime, and they're generally more comfortable with character-based UIs and the back-and-forth of a terminal session. The CLI-first workflow of modern AI coding tools feels familiar to someone who grew up with vi and grep, less so to someone whose entire career has been in graphical IDEs with drag-and-drop scaffolding.

Sunk cost fallacy and the willingness to start over

Especially early in the lifespan of a project, a lot of the work is basically throwaway. It's been like that forever, but the things you ought to have thrown away will haunt you for the rest of your lifetime. How it usually turns out is that the next sprint brings new features, not rewrites of the old ones or the long-awaited pause to think things through properly.

So what usually happens is this:

"There's never time to do it right but there's always time to do it over."
"Temporary solutions are always permanent"
"There will never be proper time to refactor"
"The release train needs to keep moving"
"Technical debt is forever"

Often it is hard to admit that we need to start over. You know, the **sunk cost fallacy** and the reluctance to admit we went wrong are both very much human nature.

I'd argue that this time it's different: regenerating major parts is a viable strategy, especially if you have good specs and know what you're doing and you can manage it. At least it's possible much later in the project than before; the attempt that produced 40,000 lines of throwaway slop might not be the disaster it would've been in the old days. Refine the architecture and your designs, and let the agents take another shot at it. At least it should be; you didn't write it yourself, did you, so what's the harm in trying again?

A word of caution though: this applies best to bounded components and early-stage work. Once the system is in production, integrated with other services, and storing real data, the window for

wholesale regeneration closes fast. [Chapter 11](#) explores that "point of no return" in detail. But within bounded scope, the willingness to throw away and redo is a genuine advantage of this way of working.

By adjusting your mental model to better **admit defeat**, consider this as a learning opportunity, regroup and do it again! Better yet, if you have lots of premium tokens to burn, you can even experiment with different approaches and see which one works best.

! Organizations that ignore this dimension will lose good people. Not because governed AI development is bad, but because they failed to recognize that they were fundamentally changing the job, and that not everyone will want the new version of it. I encourage honest conversations about what the roles will be, at both team and individual level, and offering options for people who prefer a different kind of work.

7

The Skills That Don't Come from Coding

Top survival skills for the Software Factory

The Skill Gap

I'm probably stating the obvious here: teams without training see lower productivity gains from AI tools than the trained ones. The gains might even be negative, if the first three weeks of the project are spent running 10 agents ralph-looping in-line specifications in parallel and creating a mess of conflicting plans and code.

But training for *what*, exactly? Let's take a look at this new skill set case-by-case.

Code writing

Yes, the art. The thing we used to be proud of. We knew the syntax by heart, all the APIs and patterns, all the nice principles and practices, like DRY, SOLID and so forth, and sometimes even lived by it (at least when reviewing PRs). What's aheer?

Traditional	Express intent through implementation. You think it, you type it, you debug it.
AI-Governed	Define intent through specifications. The AI types it. You still debug it.
Training	Progressive refinement exercises

Specification writing

Most developers have always hated this part, both the writing and reading. Agile gave us a brief (?) respite, you know, working code over and so forth. It's not going to cut anymore, although who really writes the specifications is another question.

So, from user stories to detailed specs will go like this:

Traditional	Brief user stories. "As a user, I want to filter the list." Ship it.
AI-Governed	Precise specs with edge cases, performance bounds, and enough context for an agent that has the memory of a goldfish.
Training	Spec reviews, spec-to-implementation prediction, examples, trials

Important addition for the specifications itself is to have a 'task template' for AI to process them. Whether it's baked in the 'User Story 2.0 for AI' or as AI documentatoin file is not important, but have a good list of things that need to be refined based on the specicication, like cross-cutting concerns, links to supplementatal material like images and wireframes helps a lot. Good news is that... you can use AI to help you with this.

Reviewing AI Output

Reviewing whatever AI throws you at from the depths of the software factory is hard just because of the insane volume it generates.

It's way easier if you manage to constain and structure the outputs, such as code, tests and planning documents coherently with project-wide practices like naming, associations. You'll need to babysit your agents a lot though that they actually follow these practices.

Traditional	Infer the author's intent from their implementation choices. Ask them in Slack if confused.
AI-Governed	Verify against spec. Detect AI pattern deviations. You can't ask the AI why (or be prepared it might 'lie' to you).
Training	Pattern deviation detection, planted error exercises, how to automate review checklists,

Architecture Judgment

Finally, the architecture. How your software is constructed from layers (onions), modules and so forth. How will it be compiled, deployed and hosted. In agile era much of this was emergent, i.e. we

started small, expanded and refactored. That can still be done, but you need to specify in much greater detail, for example how the code should be laid out. And again, erect lots of checks and guardrails to actually enforce these practices.

Traditional	Emerges through upfront design and iterative refinement over sprints.
AI-Governed	Needs to be defined clearly in text and constantly maintained there.
Training	Architecture review katas with real plan artifacts; generate something and see if it follows your general architecture.

The dominant training model today is **tool training**: how to use Cursor, how to prompt Claude Code, how to configure Copilot. Now the trend is shifting towards tuning those tools, and taking use of agents. **Tool expertise matters, but it's the easy part.**

The **hard part**. i.e. taking the game to 'next level' is teaching people to work differently and constantly improve. For many it might require developing a different relationship to the work itself.

Pick the right battles and Antti's Golden Rule of AI

Before we dive into the specifics of the skills listed above, there's a foundational instinct you need to develop: knowing what kind of problem to throw at an AI in the first place. LLMs are remarkably good at generating structured output: code, tools, scripts, transformations. They are remarkably bad, or at least rather unreliable, at being the tool themselves.

Examples of these **DOs** and **DON'Ts** below:

Don't	Load a 10,000-row CSV into an LLM and ask "what's the average salary?". It's just an expensive, slow and unreliable calculator.
Do	Paste a few sample lines and ask the AI to generate a script that computes the answer. Now you have a tool you can run, verify, and reuse.

Don't Ask AI to shift a heading 50 pixels to the left. Open your browser dev tools and do it in three seconds.

Do Ask AI when you need to restructure your entire CSS grid system and want to understand the implications.

Don't Ask AI to keep track of what you've done across sessions. It will forget, hallucinate, or lose context.

Do Ask AI to build you a CLI tool that tracks what you've done. Now you have something deterministic that actually works.

Luckily, many AI tools often resort to making Python scripts (for everything) when you feed them too much data or ask them to solve things like mathematical calculations. Yeah, sometimes but not always. Roughly speaking, remember that **LLMs are not data processors**, but they might pretend to be. By nature, and their internal system prompts, they lack the human laziness of refusing to do something too laborious or difficult for them to do. This tendency of **over-confidence and never admitting error** is often missed by humans and misunderstood as if the answer you got was actually correct.

All in all, this instinctive method of asking AI to **create the tools** rather than **solving the problem directly** is perhaps the single most important habit to develop early for any usage of AI beyond simplistic text processing.

So Antti's Golden Rule of Using AI shall be:

! Use AI to build tools, not to be the tool. Anything you can do with CLI should still be done with CLI.

Thanks to one of my brilliant colleagues for this CLI metaphor. I'm gonna use it often

Next let's head back to our Skill-writing class.

Writing Specifications for AI

The principle from [Chapter 2](#) applies here with a twist: the quality of your output is still bounded by the quality of your input, but now the consumer of that input is a machine that doesn't really think, is mostly unaware of its own limitations, and pretends a lot. [Chapter 13](#) explores the economics of specification quality in detail, but in short, the skill of writing good specifications is the **skill of knowing what the AI needs to know**, and how to say it clearly enough that there's no room for creative interpretation.

A short 101 could be like this:

Don't	Create a todo app
Do	Plan a todo app

or

Don't	Add a filter to my list
Do	Add a from-to date filter to my TasksList.tsx. Use the similar date filter as in the other grids. Limit the maximum span to one week. Use a calendar picker with manual text input. If the filter returns zero results, show a message "No tasks found for the selected date range".

To complicate things further, your prompt (or a PRD md file that you pass as context) is not the only source of information; for instance above, the standard behaviour of date filters should be in your ui-patterns.md or similar, which should be included automatically into your agentic UI developer context.

Unfortunately, specification writing is not a skill most developers have been trained in. It's a **lost art** for many. Agile's emphasis on user stories and acceptance criteria produced a generation of developers comfortable with brief, conversational descriptions of intent. You know, "As a user, I want to filter the list so I can find relevant items" is an adequate user story. You didn't need to say which list, and the developer would probably ask for more details when needed.

The brief and good-looking stories are (sometimes) disastrous specifications for AI Agents. The agent needs to know: which list? which filter criteria? which UI component? what happens when the

filter returns zero results? what's the performance expectation for 10,000 items?

Learn how to turn agents into interrogators

One of the most remarkable production boosts for generative AI is the ability to summarize and analyze text. This applies to reviewing and improving specifications as well.

So, instead of dreading being turned into a full-time spec writer instead of a coder, I have good news for you. Use the AI to interrogate you for the facts that are required to narrow the scope and what kind of holes and conflicts your story has. This kind of iterative, top-down approach to build the instructions is highly recommended, and part of for instance the GSD process I've referred to many places in this book.

Here's a sample interview by CSD planner agent when I was building another (but nowhere as interesting as the roguelike) tool, called `ctxl`, short for 'Context Lightning'. Anyway, as I was iterating a plan for the next generation of my `ctxl` tool and planning for the next iteration, the agent asked me a lot of questions about the details of what I really wanted.

It was no less ambitious than trying to optimize the context usage of an LLM and to replace or supplement the standard `read/grep` tools. Still in beta.

► Example

```
Should ctxl_symbol_body include surrounding context (e.g. the impl block a method belongs to, or the class a method is inside)?  
[1]. Symbol only. Just the symbol's own source code. Minimal, focused.  
[2]. With parent context Include the enclosing struct/class/impl signature (not body) for orientation.  
[3]. You decide: AI picks based on what's most useful for editing workflows  
[4]. Type something.  
[5]. Chat about this
```

So, no matter if you create your own to match your project or use an existing one, I recommend having a **dedicated planning or research agent** ask you these questions before you're (oh well) allowed to code anything. Given the correct recipe, LLMs are very good in finding edge cases and holes in your story as good detectives or prosecutors.

Teaching specification writing requires exercises in precision, not just process knowledge. It's an art on its own and you need to adapt it to this new way of working, too. The new specs '*User stories 2.0*' need to be tailored for the AI, but still readable by various stakeholders.

@ I never sat down to write a complete spec up front for my game. Each feature started as a rough idea, and the spec emerged through rounds of dialogue with the research and planning agents. They would point out things I hadn't considered, I'd refine the scope, and by the end we had something precise enough to execute. The specification was a byproduct of the conversation, not a document I authored.

So a good training for this could be practical:

1. Hand out relatively easy tasks
2. Have the team write specifications for them in different formats, for instance: 1) Classic user story with acceptance criteria or 2) A detailed user story many more edge cases, detailed definition of done, DON'Ts and WHAT IFs iterated and refined through dialogue with the AI.
3. Arrange a demo to run the agent create implementation and tests on these two versions of the spec.
4. Compare the results.

This will make the difference in specification quality tangible.

There's no "magic standard" for specifications, but there are definitely bad ones. I've found that having the AI produce most of the behavioural stuff as Gherkin, and the user-facing stuff as a separate design plan, helps a lot to cut down the inevitable back-and-forths with the agents.

What we actually learned

When we introduced governed AI development to a real project team, the ramp-up was slower than I expected. To be fair, much of that was the new domain and tech stack, not the AI workflow itself. Getting the workflow stable enough to trust also took its own sweet time. And there was a lot of variety among people: some jumped on board quickly, others had more trouble adjusting.

People found very different ways to cope. Some were very careful, trying to read everything the AI produced and manually approving every single tool call. Thorough, yes. Sustainable, not so much. You can imagine how that scales when the agent wants to make 40 file edits. Others went the opposite way: less reading, more trusting, full speed ahead. We ended up with totally wrong features being built because nobody had checked whether the agent actually understood the requirement or just confidently produced something that looked right.

We ran several walkthroughs, a dedicated training day, and handed out plenty of material to read. Looking back, I think the best way to learn this stuff is just to use it and become an AI whisperer through practice rather than lectures. We started with real project work instead of made-up exercises, which got us moving fast but meant people hit the hard lessons on production code. In hindsight, some controlled exercises first might have been a gentler start.

We also learned the hard way that putting several people on a relatively small codebase with AI agents is a recipe for merge chaos. Everyone's agents were happily producing code in parallel, blissfully unaware of each other's existence. The first few weeks were a festival of merge conflicts before we figured out that we needed more independent work streams.

The biggest lesson: more hands-on, 1-on-1 pairing would have helped the people who struggled. A training day gives you the concepts. Sitting next to someone while they wrestle with a stubborn agent gives you the instincts.

Review techniques

Reviewing AI-generated specifications, code, test cases and data models is a whole other ballgame from the old code-review rubberstampings.

When reviewing code written by a living human being, you can usually infer the author's intent from their implementation choices. Like, why they used this pattern, what edge case they were handling, what assumption they were making. And if required, you can comment and ask and chances are you're already familiar with domain, architecture and practices for the codebase.

When reviewing mostly machine-generated stuff, you might not have that context. And the amount of material is often just too much for anybody to actually review.

- i** AI-generated commit messages and PR descriptions are often very good, and they can be a great help to understand the intent behind the code, which is a crucial part of the review process. So make sure to read them carefully.

This changes what you're looking for. In the old world you could be semi-confident that the developer hasn't added too many extra features or interpreted the requirements entirely incorrectly. He might have even tested them before committing with a critical eye.

Don't trust the formatting

With AI, those convincing and nicely organized, good-looking documents with lots of checkboxes and colors, and the beautifully indented and formatted code might be just slop.

You've probably seen those summary messages: "All you ever wanted has now been delivered and ready for production. All code compiles and test coverage is 100%. Great success". And when you say 'npm compile' the first thing you notice is an insane number of linter errors, and the 100% test coverage was just fitted to code, not to the original requirements, and even those don't pass.

When you scratch the surface, the result you got is not doing even remotely what it was supposed to do.

Always be ready to start over.

It's pointless to argue with AI and insults won't work either. If the internal task list of your agent has 12 steps, you're at phase 7, and you find yourself in an infinite loop of same errors, and the context usage is in the red, it's time to stop, start a new session, and start over. You were just suffering from the context overload, rot, pollution or whatever, or didn't have a clear enough intent to start with. Rethink, check where the things started to deteriorate, and start a new session with a clearer plan and instructions.

In the end, not much was lost, and reassuring yourself that 'AI' will get it if you keep insisting 'I already told you you're executing the tool in wrong directory' won't necessarily help.

All in all my take-home top list of skills to learn is this:

Practice

a lot and learn when the AI is lying to you and when it's not. This is a skill that can be developed.

Don't trust

the formatting, the structure, the colors, the checkboxes. They are just a facade. Test content against the original requirements.

Use the intermediate results

as a guide to understand intent. It might be a PLAN.md, commits made, the 'Tasks' list of your agent or whatever. Did it really make sense?

Too much to read = you tried to do too much in a single step.

Break down the work into smaller chunks.

Use a separate review agent

, Pick a different model than the one that generated the code or document to have a real second opinion.

Add a new session

Press the +. The 15-30 minutes you wasted arguing with AI are gone forever.

8

Why Organizations Resist (and How They Adapt)

The Learning Curve

Met someone claiming to have years of experience agentizing their software development? Take it with a grain of salt. Frankly, the early days weren't that great after all, and it is only very recently (as of Feb 2026) we have the capabilities and tooling to seriously automate the software development lifecycle beyond the "code generation" phase, or even that.

In the previous two chapters, we looked at how individual roles are shifting and what new skills engineers need. This chapter is about the hurdles we have when trying to "Aldustrialize" our software development process.

First of all, the developers. At least many on my radar, including myself, are quite picky about the tools they use, and prefer tuning their own workflows and tools rather than going with the company standard. Many are also rather active in finding new tools and adopting (and ditching) them quickly. Also organizations, as it turns out, have their own immune system.

Resistance to AI-assisted development lives at multiple levels. Some of it is deeply **personal**, as if a developer's professional identity is tied to writing code by hand with the favorite tools. Another bar is about **competence and training**: people or team simply don't know where to start. The friction might be also **technical**, as the tools genuinely aren't mature enough for their context, sometimes leading to bad first impressions (which never go away, do they?). Finally, some (or, most) the resistance is about **organization** and politics. You know, red tape, budget, politics, rules, MORDACs. et al.

The Resistance Landscape

Where anti-AI sentiment lives in organizations

Individual

Organizational

Personal

Individual fears and identity

Fear of obsolescence

Identity tied to coding craft

"AI slop" skepticism

Loss of creative satisfaction

Distrust of probabilistic tools

Competence

Skills and knowledge gaps

No training budget or time

Bad early tool experiences

Don't know where to start

Skills gap: specs over code

Prompt engineering learning curve

Technical

Tools, quality, and integration

Code quality and tech debt fears

Hallucinations and unreliable output

Immature, rapidly changing tooling

Integration with existing workflows

Security and IP exposure concerns

Organizational

Process, budget, and politics

Red tape and compliance blocking

No license budget approved

Multi-team speed synchronization

Bureaucratic process overhead

Top-down mandate without buy-in

Most organizations face resistance across all four dimensions simultaneously

Resistance spans from individual concerns to organizational barriers

Who is pulling the brakes and why?

I don't want to make fun of sceptics here, I'm one myself. It's more than justified, especially as hype towards 'all code is written by AI' gets more steam.

Not all resistance is irrational, some luddite-style 'it was better before' principled stuff. That would be underestimating your probably bright colleagues.

So enough with my disclaimers, let's name the suspects and their motives.

The burned team

These are the teams that went through Agile transformations, SAFe rollouts, or DevOps revolutions. Many of them have the scars to prove it. They've seen "the next big thing" before, what ever it was, from monorepos to BDDs/TDDs promised to solve most or all of your problems. The pitch always sounded great; the reality was months of disrupted productivity, abandoned tooling, and a process that turned out to be more about compliance theater than actual improvement.

Much of the above is not even tongue-in-cheek, but what is perhaps very typical of our industry: what ever (usually just one thing) emerges as the Next Big Things appears on Innovative Lead's Powerpoint Stragies the next day.

So now, when you (or I) show up with AI-assisted development, they hear the same pitch.

The key with burned teams is honesty. **Don't oversell.** Start with a problem they actually have, show a small win, and let them set the pace.

The inertial team

I.e. the guys who believe 'if something ain't broke, don't fix it'.

These are the teams that are productive and comfortable. They ship reliably, their processes work, and they have no burning pain that AI promises to solve.

They might and often do have the basic 'code completions'-style AI tools in use, but they prefer to keep them at that level.

This is the hardest resistance to overcome because it's entirely rational from their current viewpoint. If our mission is to change their job descriptions from coders to agent controllers that is of course.

How to convince these guys that we aren't gonna be able to push this djinn back in the bottle, and that the competitive field is shifting under them? That you might do OK for a year, perhaps two?

I guess patience and showing results is the best bet. Have that 'AI Whisperer' to setup some nice not-too-ambitious helpers for their biggest time eaters (EVERYBODY has them). Seeing is believing.

The mandated team

Yeah there are teams that actually do something because to CTO or a similar PHB tells them to. Sounds good, right?

In my experience these are the guys who will answer "Yes" to the corporate questionnaire "Have you adopted AI tools in your software development process?" but in reality have just filled some licence order form but never installed the damned thing.

Full disclosure: I admit doing something like this in distant past myself.

The key with mandated teams is to convert the mandate into agency. Instead of "you must use AI tools," try "here's a budget and time to figure out how AI tools could help your work.". And if that does not work, send the same **AI Whisperer** the Inertial Team just laughed out with 'come back next year when you have something working'.

Wasn't the self-governing teams once the basis of productivity?

The resistance patterns

Now that I've named the suspects, let's look at the detailed patterns of resistance that they (and others) might exhibit.

The red tape

First, and often the biggest obstacle, is the red tape. The outright refusal or artificial limitation of AI tools to be used to generate software. This reminds me of the early days of web, when putting anything online was considered a major risk. News of breaks and security snafus, even recently, hasn't exactly paved the way to use AI. In the end, cutting the red tape means getting somebody higher up to sign off.

Is this really connected to internet?

There are several reasons for having governance around any IT tools, internet or not. Many of them are entirely valid in the AI era as well, such as those concerning privacy, security or IPRs. Regarding the AI tools, however, many of these concerns are being used to justify redtaping the AI tools entirely, due to a basic misunderstanding of what the AI tools really do in the context of software engineering.

Code is not Data

First of these is the misconception between the code and data. Producing sourcecode is not the same as running that code. Believe it or not, this is not a rare misconception; while running AI as part

of your runtime might be scary, and sometimes also risky, that's not what we're doing with the AI-assisted development.

Equation 1.

CODE ≠ DATA

Microsoft and Amazon Already Have Your Source Codes

Another aspect is the sensitivity of the code itself. Indeed, much of the programs that run the world are not really open source to this day. Either the area is so niche that nobody would be interested anyway, or it has some genuine inventions or trade secrets. Or, reveal the annoying algorithms to sell you your airplane ticket twice (you know, the cheapest ticket does not include right to sit, sleep or drink, or carry anything including clothes etc).

This brings me to my second equation, directly deducible from first principles:

Equation 2.

*AI ASSISTED DEVELOPMENT != YOUR CODE GOES
OPENSOURCE OR TO MICROSOFT*

If your fear is really about trade secrets, you shouldn't be running it in the cloud either, right? Or host it in Azure DevOps or github, no matter what your privacy settings are. And, there's an option to run your own in your private cloud or even on-prem, if you really want to. But the point is that the code is not going anywhere, and **the owners of AI tools are not stealing your code**, at least no more than they do already.

i I'm not arguing that AI tools should not go through an audit and due diligence process before rolling them out. Right now it seems that sometimes it takes literally years to even start the audit or DD.

We don't know how to use it and training budget is not there

Adopting new technology, especially as disruptive (there you go, I used the word!) as AI is naturally approached with caution. The field is still immature; especially the tooling scene is a mess and many probably already had some bad experiences with the early tools.

You might heard some of the following excuses:



So, "let's wait until things settle down", right?

Or, "we need the new AI guy willing to walk the walk to get the thing rolling" (and then the rest of the team will follow).

"No licence budget"

"No training budget"

"We need to get IT and Cyber involved"

"We need to focus on the current roadmap"

"They will learn by doing"

Usually the **obstacle is not the team**.

This kind of resistance or friction easily becomes a self-fulfilling prophecy.

Generally the best way to build competence is to start learning by doing. Take the first step, or the next. Get those licences out, perhaps for a small project or isolated into containers, or whatever. We and the rest of the world aren't going to wait for you to catch up in 2027-2030.

The skeptic inertia

Often the randomness and bad early experiences, combined with a real fear of losing one's job gives rise to the skeptic movement. You know, **AI slop, Who is gonna maintain this, Technical debt**. I've thrown these concerns around in the book as well, but they are often used as a blanket excuse to avoid even trying.

I'm not saying these concerns aren't valid. They are. But they are also often based on a lack of understanding of what modern tools can do, and how to use them effectively.

Perhaps the best way to overcome this class of skepticism is to show results. Start with a small pilot project, measure the impact, and share the success stories. Do it honestly and openly.

i If you are concerned about your job security as a developer: there's a good chance the total amount of work remains the same, it's just going to be redistributed. With proper use of AI tools, you can do **WAY** more and things you would have never done before. Rethink your position a bit; somebody will need to verify the code still, even debug it, and guide the AI to refactor when needed, and it's not gonna be the former UX designer doing it via AI either (to be clear, AI aren't gonna turn engineers like me as competent UX designers either).

Your expertise is not just about frameworks, programming languages and libraries.

Synchronizing the speed of change

Supposedly we can now deliver things all the way to production. This will happen at much higher speed than we used to. This might not feel like a problem at first glance, and sometimes it's actually great.

But then we enter the SAFe world (pun intended) of multiple teams, systems and dependencies, and a large backlog that just cannot be fed to end users piecemeal. You cannot install at will. It might not matter if you delivered in a week instead of a month, if you will end up just waiting for somebody else. So basically, the lead time from the idea to the end user doesn't really change much.

This brings me to *Equation 3*.

Equation 3.

AI ASSISTED DEVELOPMENT = ONLY PART OF THE VALUE CHAIN

So in a complex system, speeding up one part might not have that much effect on the lead time, if it wasn't the real bottleneck in the first place.

Practical issue related to this speed is also between the developers and stakeholders. In order to get good results from AI, you need to write and read an order of magnitude more documentation. Chances are people are busy already and cannot just bend to this insane pace and volume.

There's a more immediate price, too. You can speed things up dramatically, but unless special care is taken to keep work separate, you'll drown in merge conflicts. Multiple agents (or agent sessions) touching overlapping files at high speed is a recipe for integration pain. The architecture chapter ([Chapter 11](#)) argues for separation of concerns partly for this reason: modules with clear boundaries can be worked on in parallel without stepping on each other.

The bureaucracy issue

Going full-on specification driven with all the hooplas of the agents, reviews, traceability and all that is a lot of work and certainly something people might not be ready for.

Hell = 'French bureaucracy with German flexibility'

Let's not add any more complexity to the process if we don't have to. This is supposed to be easier, right?

The thing to consider, for which I don't have an answer yet, is to define something that is good enough for the project and the team. Yes, we need to see we're heading the right way, and yes, you're gonna need to write specs and not just code.

My solution so far is to have much of the 'good' of the old world in place. Like kanban boards, some kind of ordering of the backlog, perhaps a higher level roadmap. Use that as a source of truth, and try to ingest as much as possible from the internal loop i.e. what happens between the 'Definition of Ready' and 'Definition of Done' near the code and, consequently, the AI agents. Find or develop an easy to use tool to help you manage that.

The adoption ladder

Easiest thing to alienate your developers and screw the opening ceremony of your Agentic Software Factory is to go from 0 to 100 (kph) overnight. Take one step at a time. For example, start from the end of the process by automating reviews. Then, launch a todo-helper or a planner. And so on.

Level	What You Add	What You Get
1. Quality Gates	Manual review before merging AI-generated code	Catches the worst problems; costs almost nothing
2. Plan Artifacts	A brief plan before the AI writes code: what to build, acceptance criteria	Makes AI dramatically more effective; takes ten minutes
3. Traceability	Link plan to code to review; trace back when things break	Patterns emerge in what goes wrong and why
4. Quality Signals	Automated checks: tests, linting, no regressions	Data on how well the AI-assisted process actually works

From resistance to governed response

Perhaps all new technology will inevitably bring up it's own generation of luddites and sceptics.

They will always adapt, over time.

The patterns (however synthetic they might feel) I've described in this Chapter are not products of my imagination, but something real. Many of them I've witnessed either first hand or via community lore, tales and linked in posts. Generally speaking, I think it's always useful to think about why people think or act the way they do first if you try to change their minds. Or if you're on a mission like I'm with this book, **help them to get best out this wonderful yet quirky technology** we now have.

To quote my late ancestor, "Never go to sleep angry. I'm not saying I've been able to do that, but I suggest you try."

To conclude, here's my take on how to soften the resistance, convince the skeptics, and get the ball rolling before feeding them the corporate governed lifecycle that awaits you in the next part of this book.

- ! ***Don't wait for the perfect tool or the perfect process.*** Start with something small
- Show results.*** Measure the impact and share the success stories.
- Educate and train your team.*** The more they understand the tools and their potential, the more likely they are to embrace them.
- Address concerns and limitations openly and honestly.*** Don't dismiss them.
- Be patient.*** Change takes time, and there will be bumps along the way.

9

Accountability Over Speed

The goal is not maximum automation. The goal is repeatable delivery with accountability.

The wrong question and the right one

The dominant narrative around AI-assisted development is about going faster. But speed without direction is just expensive wandering. The same trap caught many Agile adoptions: teams interpreted "working software over thorough documentation" as permission to skip specification entirely, which was based on a false assumption of infinite budget, the possibility of redoing everything, and endless customer patience. None of these really ever existed.

To be clear, Agile's core principles (iteration, feedback loops, self-governing teams) remain sound and this book builds on them. The problem was never the Manifesto; it was how many teams practiced it, treating speed as a substitute for direction rather than a consequence of good direction. Spec-driven development restores what was lost in that translation, not what Agile intended to discard.

Going fast in the wrong direction is worse than going slowly in the right one. Or in practice, using up your premium token quota to produce code that is toxic waste, and then the next day's quota to fix it or redo it is not a good working method.

- ✓ Ask not how do we make AI write code faster.
Ask how do we make AI-assisted delivery repeatable and accountable when the whole team depends on it?

I have a cautionary tale. I created a proof-of-concept that was vibe-coded to fit the design: fast, impressive, got the point across. Later, when evaluating whether it was any good for the real thing, it turned out to be a patchwork of odd choices with almost every anti-pattern in the book present. God components, big balls of mud, the full curriculum of the vibe coding academy. It was also not tracked anywhere (a classic agile thing, right) so figuring afterwards out what features were actually

in there required some footwork. It wasn't useless, but I reckon it would have been less effort in total if done properly from the day 1.

I suspect this happens often. Vibe-coded prototypes give **initially an unrealistically optimistic image of quality and speed**. When things start to slow down (or, deteriorate if you will) to endless **refactoring loops** and **subtle bug hunts**, your burndown chart starts to look like the EKG of a cardiac arrest. Call it the "shiny prototype syndrome" if you like, but all in all it's just a healthy reminder of the fact that the **last 20% of work will require 80% of the effort**, and a healthy expectation is that with AI you can expect something similar. *Winning a battle does not mean winning the war.*

Generally, the reality of vibe coded codebases becomes apparent after a while, when somebody else (who knows *things*) takes a look, or when you yourself need to later fix or expand it. Even the best of the best of AIs will struggle with the mess.

Enforcement

So how do we build delivery that actually works the same way twice and leaves a trail you can follow? My suggestion: let's go back to basic principles of engineering: **Divide the problem into smaller subproblems until you can solve them**. To make this more tangible, let's imagine a multi-agent software delivery loop, where each AI agent owns a single stage of the lifecycle. Or a set of parallel agents working on the same stage but different tasks. How do we prevent them from wandering off into speculative work, or producing code that doesn't meet the requirements, or modifying tests to fit the code?

With CONTROL.

Enforcement Mechanisms



Three control mechanisms make this work:

Gates are enforced transitions between stages. An agent cannot advance from planning to implementation, or from implementation to testing, without meeting defined criteria. Say, the engineering agent can't start coding until its plan has been reviewed and its task list matches the acceptance criteria.

Tracking artifacts are the persistent records of what was planned, built, tested, and approved. They live near the code, in version control, not somewhere in the agent's chat history that vanishes when you close the session. Think a `PLAN.md` that links to the backlog item, commit messages that reference the task ID, and test results tied back to acceptance criteria.

Human checkpoints are hard stops where automation pauses for human approval. Not optional suggestions, but mandatory gates that cannot advance without a person signing off. You review the plan before coding starts, and you review the code before it gets merged. The agent waits.

Here's what it looks like in practice. A developer picks up a story: "Add date filtering to the task list." The planning agent produces a plan with tasks, references to the UI pattern library, and Gherkin acceptance criteria. The developer reviews the plan, catches that the agent missed the "no results" empty state, and approves the corrected version. Only then does the engineering agent start coding. When it's done, it produces a PR linked to the story. The testing agent runs the acceptance criteria. If something fails, it goes back to engineering with the failing test, not forward to review. At no point did anyone skip a step or approve something they hadn't read.

Chapter 10 describes how these mechanisms combine into a concrete pipeline with stages, agent roles, and state machines. For now, the key insight is that all three work together: gates enforce the boundaries, artifacts make the work traceable, and checkpoints keep humans in the loop.

In the end, any backlog item should be traceable all the way through to code, tests, and a pull request. Without your ever so helpful and hard-working agents wandering off into speculative work or ending up in endless loops.

Let me give you some concrete examples:

1. Require your agent to **link a commit or a PR to a specific backlog item**. This is low-hanging fruit (a good practice to have, AI or not, and yet another checkmark in your BS bingo sheet).
2. Require your agent to **produce a detailed task list**, and refer to specific supporting documents and artifacts it should be working on. Like plan a task to add a new API method: link to the API specification in the task.
3. Link **tests to your source of truth**, i.e. the place where your spec-driven-design is being documented. An `.md` file with acceptance criteria, something unstructured, anything. You'll be able to later discover, why is this test here?

And so on.

What I'm suggesting here is not about replacing your judgment with AI automation, or about slowing things down with bureaucracy. It is about drawing a clear line between what the agent decides and what you decide, so neither of you wastes time on the other's job.

Why "Accountability" matters

In professional delivery, where my day job is (my roguelike is still not ready to be monetized, as you might have guessed), accountability is not a nice-to-have. Clients need to know what was built and why. Somebody might need to trace decisions. Teams need to onboard new members who can understand the history.

Without accountability, AI-assisted delivery becomes a black box: code appears, nobody can explain the decisions behind it, and the organization loses the ability to govern its own software. Well, that's an accurate description of most agile projects, right? After 20 sprints with 10 incremental changes to some feature, good luck figuring out why the code is the way it is, and what was the rationale behind it.

i Naturally, a solo developer building a side project or a research-oriented team creating a PoC does not need any of this. But for teams delivering software at scale, the tradeoff tends to pay off.

The chapters that follow describe how we have implemented these principles in practice. I describe the specific framework, its specification-driven design approach, and the governance boundaries that make it work. Treat my words as a starting point for your own thinking, not as a rigid methodology or an off-the-shelf Claude Code plugin to be installed or followed without question, customization, and considerable effort. What I really wish is that all of you who are voluntarily or involuntarily looking into this would get insights and not step on each landmine I have.

10

The Pipeline: Stages, Gates, and Artifacts

The core building blocks of your Software Factory

The ticket lifecycle

After all this handwaving, it's time to get more practical. Let's consider one unit of work: a single user story that flows through five stages. Four are agent-run; one is your existing CI/CD, agentized or not. Each of these stages has a clear intent, input and output and is probably very familiar to you from the pre-AI era.

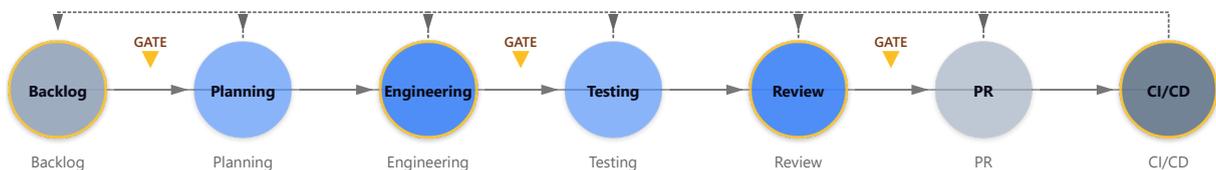
Backlog item → Plan → Engineering → Testing → Review → PR/Merge → CI/CD

Each of these arrows represents a **gate**, basically an enforced state in a state machine. Agents cannot cross gate boundaries without meeting defined criteria.

The image below depicts the core stages, gates, and artifacts in the governed lifecycle. Each stage has specific inputs, outputs, and responsibilities. The gates enforce quality and alignment before allowing progression to the next stage. Perhaps at some point, you might be able to remove human involvement via some smart automation, or require it only if a deviation is detected. (More about this in the Feedback Loops chapter).

Stages of Software Engineering with AI

End-to-end flow with gates and checkpoints



Each gate is a checkpoint where humans and automated tests validate before moving forward.

Reject-back loops allow rework when issues are found, maintaining quality throughout.

This is a good template to start building over the standard "Plan mode" and "Agent mode" once you've discovered the need for more structure and control.

Planning

In case you wonder, this is the 'outer loop' work that happens before any code is written. It sums up the requirements, designs, and whatnot, and turns them into atomic, structured, and ordered tasks that you can feed into your pipeline.

- i** Don't let the planning agent have access to the backlog item or its conversational context. It should only get the structured requirements, designs, and relevant codebase references. This forces a clean handoff and prevents context bleed.

Specialist planning agents can be spawned to create and review planning tasks when needed. For instance, if you're porting an old frontend and need to learn from it and produce designs, you might use a differently instructed agent than for a greenfield feature. Same if you have a layered architecture or microservices; some parts of the system could benefit from a domain-specific agent with specialized knowledge and context. Apply this specialization as you see fit, but remember: the smaller and more focused an agent is, the better it will perform.

For example, a good plan for an agent might include:

Plan Element	Example
Flow	Add a filter to the data table
Alternatives	Gherkin-style acceptance criteria that can be directly mapped to tests
Architecture	Implement as a new API endpoint that the frontend will call
Components	Backend service A, frontend component B
Scope	Don't change the database schema, just add a new API that returns filtered data
Task List	A detailed 'DoD' style list of tasks, to be performed in a certain order and won't take hours to complete.
References	Link to relevant codebase files, design docs, UX mockups, task-specific DoDs, if any.
Exclusions and Constraints (the leash)	"Don't create any database tables", "Don't add pagination to the API or the screen". Etc. Otherwise you'll risk getting things you didn't ask for in the first place, as AI tends to be rather creative!

- ✓ Tired of reading? Open a fresh session using another model and ask it to review the plan with a critical eye. While you're at it, craft a good review agent to do this for you. You might be surprised at what it catches.

Engineering

The engineering stage is where the rubber meets the road and used to be the one with the most attention and often work. An Engineering Agent takes the approved plan and executes it, producing code and unit tests that meet the defined acceptance criteria. This stage is all about **execution**: translating the plan into a working implementation as you planned, but nothing more.

The primary intent of the engineering agent should contain:

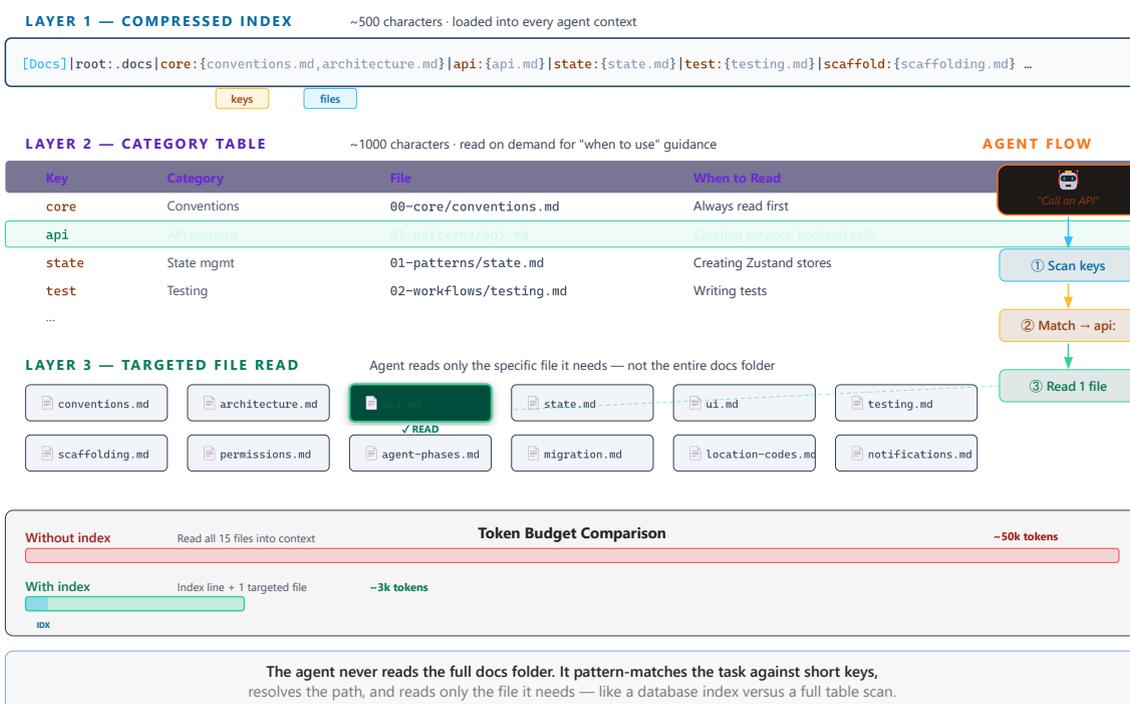
1. Have a carefully crafted context that follows your project practices and architecture
2. Execute a clear workflow that dictates the gating and required additional tasks, such as a requirement to build everything before declaring victory
3. Enforce revision control discipline
4. Do a self-review before asking for a human review (if required)

Compressed Indexes: Context Engineering for Documents

One practical tip for controlling the task-specific project context fed to agents is a technique called **Compressed Indexes**. The basic idea is that instead of chunking everything into your CLAUDE.md or maintaining separate reference sections for all agents, organize your documents into meaningful chunks by theme, backed by an accessible index. Let the agent seek advice from it when needed. It could be something like this:

Compressed Index: Agent-Friendly Documentation

Three-layer lookup — from a single line to the exact file an agent needs



A good index for LLMs is not just a table of contents; it should be designed to help the agent find the right context efficiently, without consuming unnecessary tokens.

The index should include:

- **Document summaries:** A brief description of each document's content and purpose.
- **Relevance tags:** Keywords or tags that indicate which agents or tasks the document is relevant for.
- **Sample questions:** Example queries that would lead the agent to consult that document. This helps the agent understand when to seek that context.

Check the link below for an open-source implementation of this technique.

[ai-docs-indexer](#)

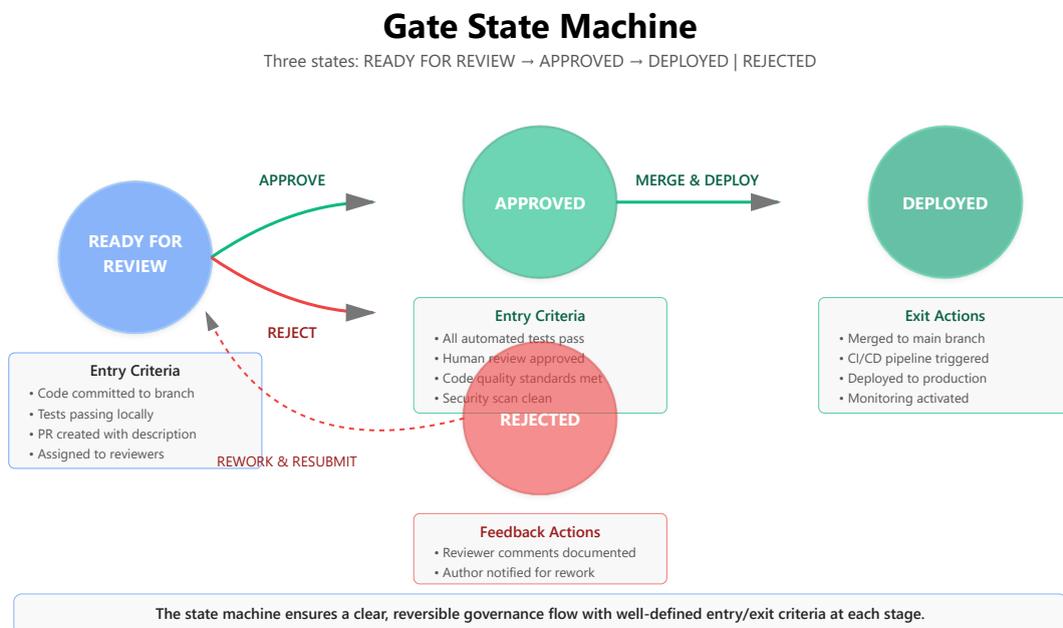
Specialist agents

As you learn along the way, you may end up having specialized engineers for different aspects of the work, like frontend and backend engineer agents. For instance, a **"critique" agent** (think of the guy in your team who is vicious about indentations (the correct answer is 3, spaces btw)) that reads your code and shreds it to pieces can also be a nice touch. Spawn it in a fresh context and with a different model before asking for a human review.

- ✓ Don't let the engineering agent have access to the planning agent's conversational context. It should only get the structured artifacts and the current codebase state. The same applies for agents writing tests: given the code, they will produce tests that pass by fitting the code, not what it is supposed to do.

The gate state machine

Here's one example of how you could design your own work tracking engine. Essentially it's the same as we had back in the day as 'Definition of Ready' and 'Definition of Done,' but with more structure and automation; when the feature is clear enough to be worked on, when the code is good enough to be reviewed, and so on.



Whatever the exact gates you erect, in the absence of humans in the inner loop, you need to have something like this to prevent runaways and to maintain traceability.

I've provided a more detailed breakdown of one possible setup that we've been using.

Agent roles and responsibilities

Every governed pipeline needs a set of specialized agent roles. The exact names and boundaries will vary with your stack, but the division of responsibility matters more than the labels. Each role should have a clear mandate, defined inputs and outputs, and explicit criteria for advancing or falling back.

Role	Mandate	Advances When	Falls Back When
Roadmapper	Transforms use cases into structured backlog items with acceptance criteria	Items meet clarity, testability, and vision alignment	Items are vague or misaligned
Planner	Turns backlog items into ordered plans with tasks and references	Plan is complete, clear, and architecture-aligned	Plan is incomplete or misaligned
Engineer	Produces code and unit tests per plan	All tasks complete, build passes	Timeout or build failure
Tester	Validates implementation against plan and acceptance criteria	All tests pass	Tests fail or are incomplete
Reviewer	Runs quality checks against a defined checklist	All checks pass	Issues found

The critical pattern is that each role consumes only the **structured artifacts** from the previous stage, never the conversational context. This prevents context bleed and keeps each agent's input clean. The reviewer, for example, checks code quality, performance implications, accessibility, functional correctness against spec, architecture alignment, and UI/UX adherence.

✓ [Chapter 12](#) walks through a concrete implementation of these roles: the specific agents, their recipe files, and the lessons from running them on a real project.

How to structure the backlog

The classic backlog, you know, the 'Epic/Feature/Story/Task' or whatever your ticketing system calls them, is a good blueprint on how to organize your work. And chances are you're required to use it. The key is to design it as hierarchical and sequential, as you'd normally do when working with flesh-and-bone developers.

And in the end, you had only one-line user stories. 100 of them in the "Active" column

Here's a setup I've found to work well in practice.

Artifact	Stage	Creator	Format	Consumer	Purpose
Feature	Planning	Roadmapper Agent	Markdown/JIRA	Backlog item	Project-level requirement
Story	Backlog	Product	Markdown/JIRA	Planning Agent	Requirements source of truth
Plan	Planning	Planning Agent	JSON/MD	Review gate + Engineering	Architecture and task breakdown

Artifact	Stage	Creator	Format	Consumer	Purpose
Tasks	Planning	Planning Agent	JSON/MD	Engineering Agent	Executable specification
Code + Tests	Engineering	Engineering Agent	PR	Testing Agent + Review Agent	Deliverable
Test Results	Testing	Testing Agent	Test Report	Review gate	Validation evidence
Review Checklist	Review	Review Agent	Structured	Deploy gate	Quality assurance
PR + CI Report	Integration	Your CI/CD	Standard	Deploy team	Production readiness

Having this systematic and well-defined is a bit like programming the workflow. Clear input and output for each stage keeps things on track. With humans, certain degrees of freedom and flexibility are expected, you know, Markku will surely know how to add this cancel button without telling much more, but the agent won't. And you can **still fix a single thing without a ticket**.

Making your backlog items AI-ready

Traditional agile stories are deliberately lightweight: a sentence of intent, a few acceptance criteria, a conversation placeholder. This works fine when the executor is a human who will ask clarifying questions and fill in the gaps from experience. It works considerably less well when the executor is a Planning Agent that fills gaps probabilistically, i.e. by guessing.

On one of my projects I found out the hard way that the standard 'As a user, I want to...' format was nowhere near enough. The planning agent would produce technically plausible but contextually wrong plans because it had no idea which APIs were available (and what kind of input they'd require and what kind of data you'd get back), what the UI conventions were, or where the architectural boundaries lived.

Naturally these kind of concerns, at least some of them might reside in project documentation, as they should, but in pre-planning stage a good 'Definition of Ready' checklist for the backlog items is a good way to make sure the planning agent has what it needs to produce a good plan, and also coaches the developer in charge of the 'production line' of the agent factory to pay attention.

Think of it as the 'pre flight checklist' for the planning agent. If you have good structure there, even with gaps and just titles, your detailed plans end up looking much more like each other and limit the creativity of the agent in a good way.

So what helped was treating the analysis phase as a proper **Definition of Ready** checklist before handing anything to the planning agent. For each story, we added:

- **Architectural guidance:** which layers and modules the feature touches, and which it must not

- **Technical analysis:** relevant existing patterns, data models, and constraints
- **Integration points:** required APIs, services, and their contracts
- **UI/UX decisions:** layouts, controls, behaviors, and the reasoning behind them
- **Acceptance criteria:** clear, testable conditions that define when the story is done

This is not a requirements document per story. It's maybe ten to fifteen minutes of extra work during analysis, but the downstream impact on plan quality was significant. The agent stopped inventing architecture and started following it.

Bugs discovered later in the process should be judged by the same criteria: if they are big enough to require a plan, they should be treated as a new backlog item and go through the same process. If they are small enough to be fixed without a ticket, then just fix them.

While the details and complexity of your dream setup will certainly be something other than what's above, remember that you need some kind of clear system to maintain traceability and control.

In summary, a good agentic lifecycle is not just about the agents and their capabilities. It's also about how you structure the work, define the stages, and enforce the gates to maintain quality and alignment. A good one lives by these principles:

- ? **Forces clarity:** Vague specs become obvious when you serialize them to a structured format and break them down into tasks
- ? **Enables automation:** Each agent can validate entry criteria before starting and refuse to work if they are not met.
- ? **Creates evidence:** Audit trails show what was decided and by whom. Enables you to go back, improve the plans and agents, and perhaps regenerate the code.
- ? **Allows rollback:** You can revert from clean artifact boundaries. The tester won't fix it but relegates the fix to engineering. Engineers can't figure out (trust me, I'm one) what the customer really wanted, so they ask the planner to clarify and update the plan. And so on.
- ? **Keeps things in check:** You know what was done and what wasn't. You can resume and validate the work at any point and maintain a clear picture of the progress and bottlenecks.

What not to do

Some things are perhaps not to be GenAI-automated due to several good reasons. I'm the first to admit I do have an 'infrastructure engineer agent' in my toolbox, whose idea really is not to act as a runtime (although it might be used to diagnose!), but to produce scripts and tools for the humans to

execute. The risk of letting it execute them is just too high, and the potential benefits are not that great.

So some words of warning of what not to agentize just yet. You may thank me later.



Don't replace CI/CD Your CI/CD practices should remain unchanged. You can and will CI much more frequently than before, leave the CD behind a manual gate.

You don't want a DevOps Agent Don't let the AI manage your infrastructure, pipelines, or deployments. It can produce handy scripts and tools for it, but don't let it execute them. Thank me later.

No DBA Agent either Don't let the AI modify your database schema or manage migrations. It can generate migration scripts, but the execution should be a human decision.

No IAC or runtime work Modifying (and probably breaking) your pipeline or the infrastructure will cause havoc. Use the AI to create handy tools for it, but don't let it execute them.

Don't manage cross-team dependencies Getting simple tasks to flow will be hard enough. Leave the project or team boundaries to your PM.

**Don't fix bad backlog items or invent new ones:* You'll be the only one to tell what is really needed and what is not.

Feedback loops that actually work

The pipeline as described above is mostly forward-flowing: backlog to plan to code to test to review to merge. Rejections loop back one stage, and that's about it. But real delivery needs richer feedback than just "send it back to engineering." I'll cover the broader feedback loop design in [Chapter 20](#), but two loops proved their worth early enough to mention here.

The first was **capturing reusable elements**. When the tester or reviewer spotted a pattern that kept recurring, a shared call pattern, a common UI control, or a cross-cutting concern, that information got fed back into the architecture context documents that agents read for future tasks. Without this, the agents reinvent the same patterns every time, and you end up fixing the same things over and over.

The second was the **revert-to-engineering loop** for bugs discovered by the testing agent. Instead of having the tester attempt fixes or the engineer debug within an already cluttered context, the bug report went back to a fresh engineering session with just the failing test and the relevant code. Concerns stayed separated, context stayed clean. Exactly the kind of thing that matters when your executor has a finite context window and no memory between sessions.



The feedback loops that actually worked were the ones that fed information *back into the system's memory*: architecture docs, shared patterns, conventions. The ones that stayed aspirational were the meta-loops: metrics calibrating the process, retrospectives tuning the governance. Start with the concrete ones.



The game eventually acquired its own lightweight version of these mechanisms: a milestone plan, per-feature specs as structured prompts, and a personal review step before each merge. No state machine, no shared tracking. Just habits that emerged from getting burned too many times. The framework in this chapter is what those habits look like when they need to scale beyond one person.

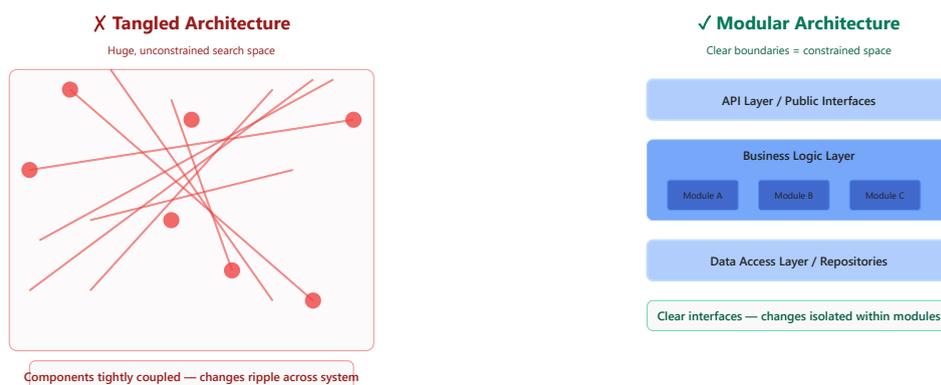
11

Architecture as Probability Management

Why architecture matters more, not less

Does the fact that more and more code is going to be AI-generated make architecture and software design less important than before? If you can **regenerate anything** at any time (or so they'll tell you), why care about its internal structure or technical details? Things are certainly going that way; CLI tools don't even show you the code unless you're interested. Like no-code/low-code, but actual custom code that you just won't see. Many are completely willing to let the AI define the higher-level architecture from a set of existing blueprints, or even let it just come up with something on its own. After all, if it doesn't work, you can just ask it to fix it, right?

Architecture Constrains Search Space



Good practices, modularity and clear separations of concern give your coding agents a living chance.

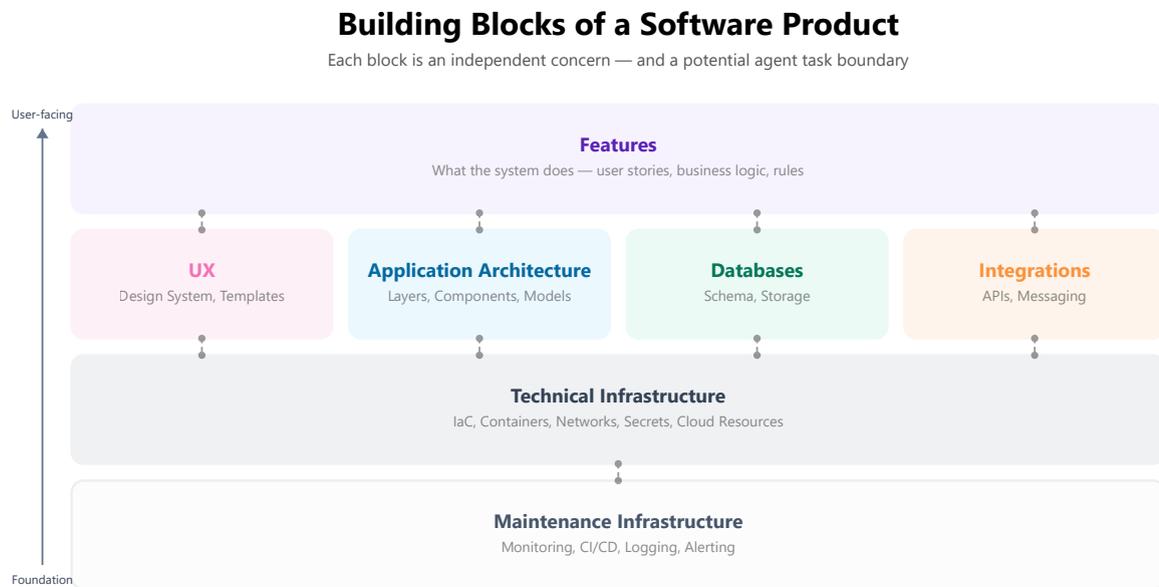
I'd like to disagree about this at least for now.

For projects with low expected lifetime, simple structure and limited feature set this might actually be reasonable. The ability to produce amazing-looking almost fully functional systems **with zero lines of code from the developer**, is of course a result of insane amounts of training data fed into models. In a way, what you've seen is Stack Overflow 2.0 in action: like it or not, you are witnessing patterns in the training data being reproduced rather than something entirely new or creative.

I'm a great fan of using established patterns, practices, blueprints and reference architectures for e.g. application architecture, infrastructure, security and so on. *That's what we've been up to since the internet became a thing.* But that's still a far cry from letting go completely of the system you might end up giving guarantees for. Or having your employer's name in the headlines with unflattering context such as "data breach" or "security incident", or just for having to maintain the thing for a long time.

So, I'd argue the opposite is true for anything more complicated than a simple CRUD web app with a hamburger menu on the top left and a persona symbol on the top right (funny how this became the standard layout at some point). Consider the figure below which illustrates the core building blocks of a software product. For the time being, I'll continue to insist on **knowing what these blocks are**

made of, and why, but will remain open for good suggestions, and have certainly no desire to write any IaC code myself anymore.



The building blocks of a software product. Each block is an independent concern with its own patterns, conventions and task boundaries.

So my point is that you need to pay **more attention to architecture**, not less; it's in no way easier to correct big upfront errors later on, AI-generated code or not. It's the other side of the 'spec-driven design': define the structure and rules for the code to be produced first, and then let the model fill in the blanks. The architecture is the specification, and the code is the implementation.

Next, I'll explain in a bit more detail why I think you should still care.

Don't bite more than you can chew

An agent working within a well-defined module with clear inputs, clear outputs, clear boundaries, clear intent or concern has the privilege of operating in a small solution space.

Basically the number of options is limited and manageable. It's like sending 007 on a mission in London **to catch a Spectre agent in the British Museum**. Yes, there's a lot of rooms to hide, but the doors are locked and the layout is familiar. The probability of success is high.

When M sends James to Istanbul, however, his **problem space gets vastly bigger**. So many streets, waterways, and people talking a foreign language. In the movie, the bad guy conveniently shows up at the casino, but in your less martini-intensive world of coding that won't happen.

So, for James working across poorly separated concerns creates an enormous solution space where the probability of a correct end-to-end solution drops exponentially. It's the same for us developers: good luck finding anything in a codebase of a few hundred kLOC with no naming standards, modules, or clear patterns.

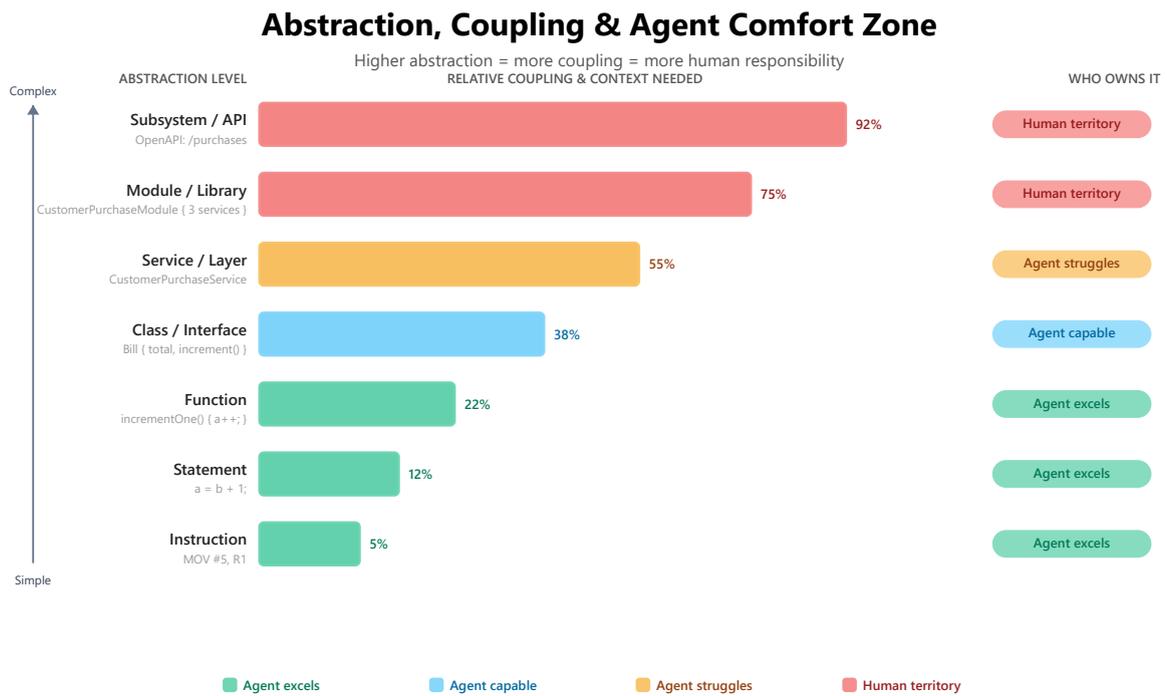
Good architecture tries to have clear patterns, boundaries and conventions that limit the solution space when developing a feature, fixing a bug, or implementing a change.

i The Step Size Principle from [Chapter 5](#) applies directly here: tasks that cross multiple modules or layers have a lower probability of success than tasks contained within a single module. So keep the layers organized.

Separation of concerns

Classic software architecture principles and patterns like *separation of concerns*, *modularity*, and *layering* (low coupling, high cohesion) are often seen as a recipe for maintainability and comprehensibility, a way to build software while remaining sane.

While I agree with Dave Farley's notion that boring code is good and that smart patterns are actually harmful, he certainly didn't mean the "100 different ways to do the same thing in the same app" kind of boring. He observed that AI tends to write "boring code": long functions with few branches, lots of static declarations, no fancy patterns. But that doesn't mean the code is well-structured or that it has clear separation of concerns, which is another thing entirely.



Each abstraction level adds coupling and context requirements. Agents thrive in constrained spaces; humans must own the boundaries.

I'm not willing to give up easily on having proper, well-thought-out, and human-understandable structure even when you let a model do the typing for you. (And don't get me wrong, Dave Farley wasn't suggesting that either.)

At some point, you will need to take the steering wheel to debug, fix, and sort things out. If what you (or the poor AI agent) need to look at is just a big ball of mud with spaghetti, **good luck finding that subtle bug.**

In AI-assisted development, separation of concerns is not just about maintainability. It's about probability management. Each layer boundary is a context boundary that limits the information an agent needs to process, which in turn limits the probability of context-related errors.

And let's not forget the quality attributes (performance, security, reliability) which you cannot test in a code review or with Playwright, or express as tasks with clear intent for agents to execute. These are baked into the architecture, which is exactly why they're called cross-cutting concerns.

Context engineering and architecture

Context engineering is typically discussed as a prompting technique, using subagents to start fresh with a smaller context, or including only the relevant information for the task at hand. But the most powerful context engineering happens at the architecture level, before any prompting begins.

A well-architected codebase can be described to an agent with a simple diagram or list: a map of modules, their responsibilities, their interfaces, and their dependencies. It does not matter if this is the CLAUDE.md file, an architecture.md, or anything, but the key thing is to specify which modules exist, what they do, and how they interact. This is the context that agents need to navigate the codebase effectively.

Here's a real example. This is a compressed documentation index from a project, designed to be loaded into agent context in a single line:

```
[Docs] | root: .docs | core: {conventions.md, architecture.md} | api: {api.md}
| auth: {authentication.md} | state: {state.md} | ts: {typescript.md}
| ui: {ui.md} | perms: {permissions.md} | scaffold: {scaffolding.md}
| test: {testing.md} | agents: {agent-phases.md} | migrate: {migration.md}
| sse: {notifications.md} | ver: {version-management.md} | skills: {reference.md}
```

The agent doesn't need to read all of these. A routing table tells it which to load on demand:

Key	Category	When to Read
core	Conventions, architecture	Always read first
api	API integration patterns	Creating services, backend calls
auth	Authentication patterns	Anything touching login, tokens, sessions
ui	UI patterns and components	Frontend work
test	Testing workflows	Writing or reviewing tests
agents	Agent phase definitions	Orchestration and pipeline work

The CLAUDE.md simply points to this index with one line: "Detailed guides in `.docs/` — see `.docs/INDEX.md` for full structure." The agent reads the index, picks the relevant docs, and ignores the rest. This is architecture serving context: the documentation structure mirrors the codebase structure, which means the agent only loads what it needs for the module it's working on.

When your system grows, you might separate, for example, the frontend and backend into different files and make sure they are loaded on demand from separate files. The microservices-backend-engineer-agent could and should not care less about your next.js or tailwind configuration and practices.

Convention files as architecture enforcement

Coding conventions aren't just style guides: they're architectural constraints expressed as rules. Some of them are related to syntax or naming, some of them are patterns. I'm fond of rules like

"Never create a new service or API without asking the developer first." to prevent slop and limit the **combinatorial explosion of new files and classes** many LLM agents are overly keen to create.

Also, rules like "Services always return Result error types." "Components never call APIs directly, they MUST use service adapters." "State mutations always go through the store." are good candidates. When these conventions are documented in files that agents read as context, they become soft enforcement of architectural boundaries.

Judging from the abundance of ALL CAPS emphasizing in this AI scene hints at something, doesn't it. Not 100% convinced they have that much effect.

It's a matter of scope and agent design when these rules are applied. I'd prefer as early as possible, meaning that the plan should already contain tasks that will follow your architecture. So when the coding starts, the agent is already in the right context and following the right patterns. Another approach is to trust the CLAUDE.md and have a strong belief it'll be followed throughout the process. In my experience, a detailed task list or todo list designed according to your separation of concerns tends to yield better results.

In practice, agents violate boundaries in predictable ways. The most common: doing exactly what was explicitly prohibited, like altering files they were told not to touch. Agents also love to stop after each step and ask for confirmation even when specifically instructed to continue autonomously, or just quietly disregard project context directions when those directions conflict with what seems "reasonable" to the model. Your best defenses are `.llmignore` files and tool restrictions rather than polite instructions. Soft rules get soft compliance.



The single most persistent violation is one-shotting. Despite explicit instructions to work one task at a time, agents will attempt to implement an entire feature or multiple tasks in a single pass. The only reliable fix I've found is limiting the scope specifically to a single task per invocation. Telling them not to one-shot doesn't work; you have to make it structurally impossible.

For example, instead of giving an agent a vague task like

► **Example**

```
'Task: Add a button and an API call to service XYZ to perform action ABC'
```

You might have a better chance of NOT violating your house rules when the task is like:

▶ Example

```
'Task 1: Add a button to the UI on Page XYZ on the top left menu to call "Do action." Wire that to serviceX/doSomething API (POST)'  
'Task 2: Add a new route to serviceX/doSomething and POST handler'  
'Task 3: Implement the API handler to parse the action and save the results to the DB.'
```

As a bonus, you don't need to write these yourself. Just make sure your plan is atomic enough.

Module-level context scoping

When an Agent works on a task, it should receive the following inputs:

- the task specification,
- the relevant module's code,
- the interfaces of adjacent modules
- the conventions document

and nothing else. Much of this is automatic and can be inferred from the task, conventions, and by traversing the codebase. Limiting the search space to the correct module, directory, or layer makes the difference between looking at a chance of 30 files edited with +1235 and -394 lines, and 4 files with +23 and -4 lines. Believe me.

So, every additional file, perhaps found by grepping for keywords in the context, is a potential source of confusion. Architecture that is 'good' for AI is modular, has clear interfaces and separation of concerns: it limits the search space. You can increase the intelligence of your coding tools by using additional indexing and graphing tools, such as Serena, to discover structures and patterns in your codebase.

↗ <https://github.com/oraios/serena> — Serena

Designing for regeneration?

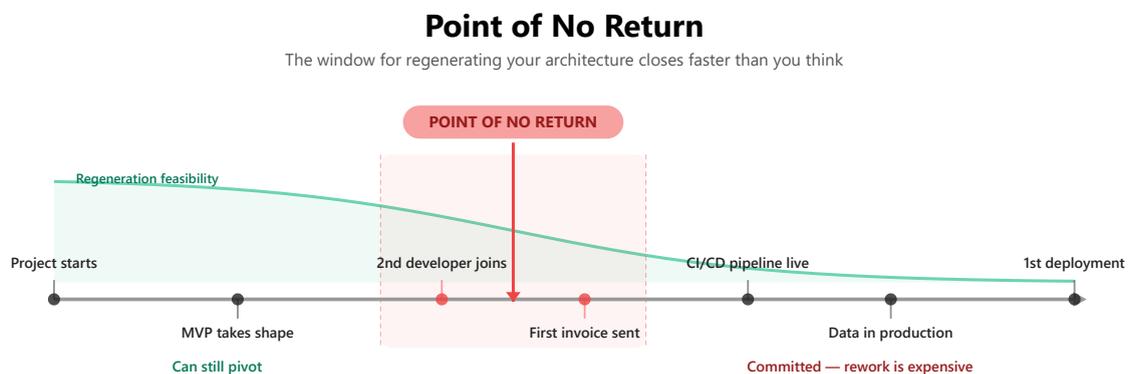
I saved this for the last.

Let's imagine you still cherish the idea of being able to regenerate major parts of your codebase at any time. Before I can talk you out of the idea by lecturing about the importance of architecture, let's consider what it would take to make that a reality.

First thing is the state of your project. Has any of it already been in production, possibly integrated into some other system, and stored some data? Your re-generation option just went out of the window.

Second, have the developers, agents, and pipelines already been somehow fitted to your initial (or 2nd, or 3rd) architecture? If so, you'll need to redo or at least verify them again.

Need I go on? Realistically, if you have more than one person on board, have already worked quite some time, and potentially spent some (customer) money on the project, **you are not going to be able to just throw it all away and start over**. Your AI tools will also struggle when trying to fix it or add new features to it. It's a major screwup, and you cannot hide behind the 'but the AI did it' excuse. I will conclude the chapter with a diagram of "Point of no return" in software development.



The window for major architectural changes closes well before your first deployment.

! Do your homework first, stick to the planning mode as long as possible and have somebody else look at the first shots before investing significant time and resources.

12

The Framework in Practice

From theory to a running system: what we actually built, how it runs, and what we learned

The previous three chapters described governed delivery in the abstract: principles, pipeline mechanics, and architecture as probability management. This chapter shows what it looks like when you build one. This is not a cleaned-up demo, but the actual system, with its ten agents, its CLI-enforced gates, its skills library, and its warts.

My intent is not to provide a template you should copy, and I'm not going to publish these MD files and bash scripts on GitHub. Your stack, your team, your domain will produce different design decisions.

The system

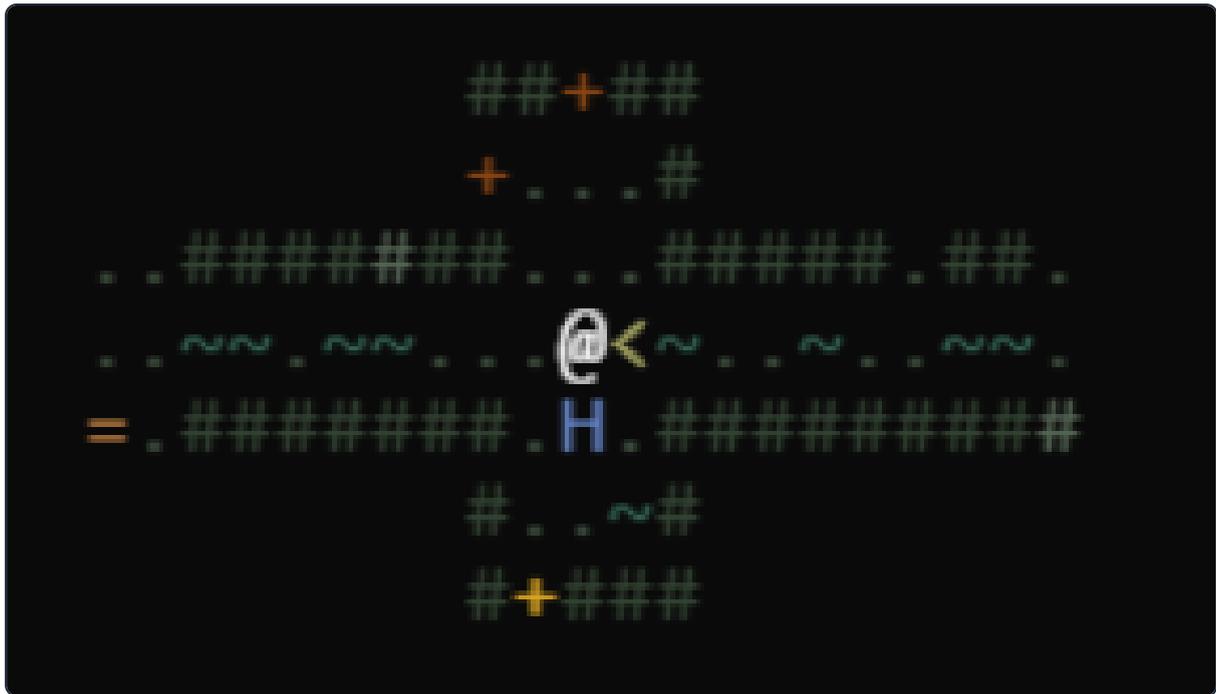
To address a new promising project on an I-horizon (and, I hate to admit, from some kind of bizarre academic interest), I set out to build a multi-agent framework to help me and my team with the new challenge. I had my background already, but was still more or less in the single-developer sandbox (or a container), where my colleagues and I indeed used AI, but with no system -- let's say ad hoc.

Originally GitHub Copilot and VS Code was our target platform, but I soon extended it to work with Claude Code (and OpenCode et al.) to remain tool-agnostic.

Despite some convergence (like for 'agent' and 'skill' 'standards'), co-living with Copilot and Claude is not really very smooth. IETF or W3C to the rescue?

This setup, which I still find growing and evolving each day, orchestrates AI agents through a gated workflow that mirrors the stages described earlier in this book. It covers the phases from requirements decomposition through planning, implementation, and testing, all the way to code review and pull requests.

Each phase is owned by a specialized agent, and transitions between phases are governed by validated gates.



A new level

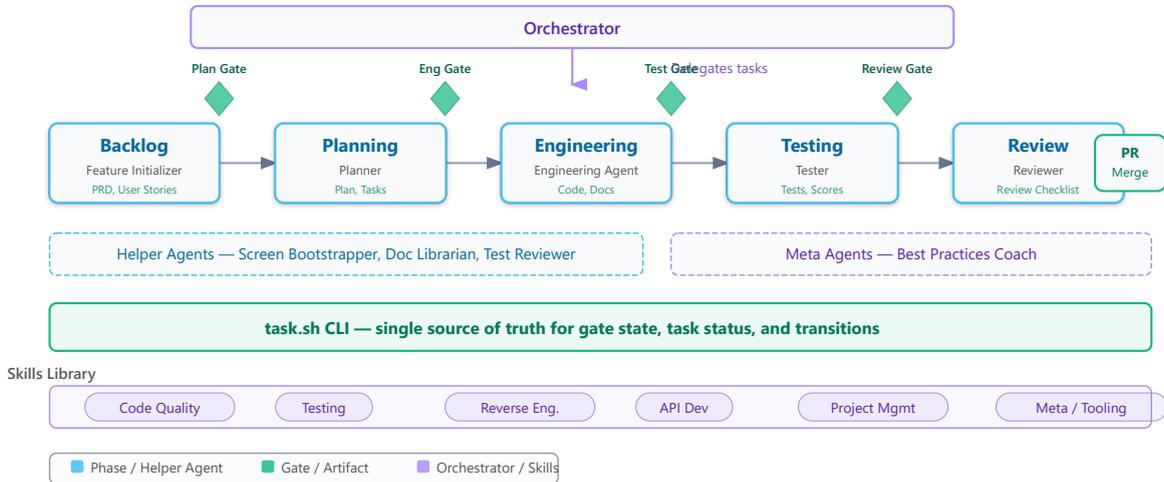
@ Taking your agentic work to the next level requires certain discipline and structure. Like in the screengrab above, where I've just *descended* into yet another level of the 1902 asylum, overrun by a group of ancient-god-worshipping cultists, and worse.

The project context that this system was designed to support was a project to rewrite a large app in a new infrastructure and modern technology. There's a ticket system as the backlog and a CI/CD system. That context shaped many of the specific choices, such as certain skills, the ticket system integration, and the screen bootstrapper agent. I've also included and tuned various skills and agent templates from public sources, which I found useful in my work.

Anyway, the structural patterns I've put into place may be used in any project where you want governed AI delivery. An overview of the setup is depicted below.

Governed Pipeline Architecture

Stages, gates, and CLI-enforced state machine



The governed pipeline: agents, gates, artifacts, and human checkpoints

Separation of concerns: Phase, helper, and meta agents

[Chapter 10](#) outlined the agent roles a governed pipeline needs: roadmapper, planner, engineer, tester, reviewer. Here's how those roles materialized in practice. The system comprises ten agents divided into three categories: **phase agents** that own the workflow stages, **helper agents** that are used as delegates by phase agents for specialized work, and **meta agents** that operate across the pipeline rather than within a specific stage, and are more about the agentic system than the software project itself.

The Orchestrator

The Orchestrator is the central coordinator of the entire pipeline. It is a stateless router that inspects the current gate, validates prerequisites, and delegates to the correct phase agent. It never executes work itself but ensures that the right agent is activated at the right time based on the state of the task tracking system and the defined workflow.

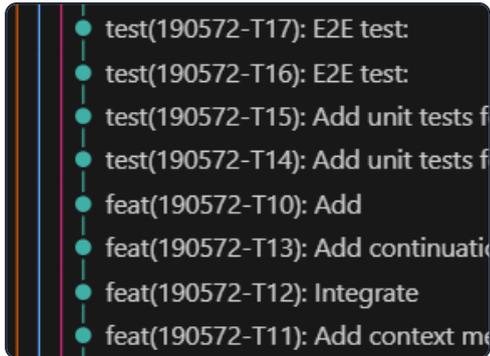
In the end, once we've done our prework for a user story, and are about to start, it should be as simple as the following to get the system going:

► Example

- > **You:** Execute feature 2534, Story 12542
- ◆ **Orchestrator:** Working... (finds the story in the 'Planning' gate)
- ◆ **Orchestrator:** (spawns the Planner agent to handle the planning phase for this story)
- ...

Phase Agents

In our setup, the phase agents are designed to run a single workflow, typically a task that can be completed on a single run. To run engineering and testing tasks, such as 'implement API endpoint', we found it to be useful to have the agent spawn a separate session for the individual, atomic tasks instead of passing the entire list of 10-15 smaller steps.



Atomic commits from the Engineering Agent's task breakdown

What I've found particularly nice about this small tasks approach is that the GIT history comes automatically rather self-descriptive, atomic and easy-to-revert, as each of the 'engineering' steps becomes a single commit.

On the left, I've provided an anonymized snapshot of a git history with a medium-sized user story with 20+ engineering and testing tasks. Each of these tasks was spawned by the **Orchestrator** to a specialist agent to run it, each with its own clean context and task specification stored in my task-tracking system.

Each phase agent owns a workflow stage for the task given by the **Orchestrator** and is responsible for driving it to completion.

Agent	What It Does
Feature Initializer	Fetches PRDs, generates user story drafts, creates tracking entries
Planner	Reads user stories, analyzes legacy code, produces implementation plan with task breakdown
Engineering	Implements tasks one by one, updating status after each; runs build validation before advancing
Tester	Designs and executes end-to-end tests based on acceptance criteria; validates test quality using a scoring rubric
Reviewer	Performs read-only code review against project standards, architecture rules, and quality metrics

Helper Agents

Helper agents are invoked by phase agents for specialized work that requires different context or instructions. So far we haven't needed that many. This might be rather easily extended once more specialization is needed. Below, I've provided a snapshot of one's I've discovered to be useful in our context.

Agent	Invoked During	What It Does
Screen Bootstrapper	Planning	Designs visual layout documents for new UI screens (specific to our frontend modernization context)
Test Design Reviewer	Testing	Scores test quality using Dave Farley's 8 Properties of Good Tests, producing a numeric "Farley Index"

-  One particularly nice helper is the Test Design Reviewer, which implements Dave Farley's 8 Properties of Good Tests and produces a numeric "Farley Index" for the test suite. I've even used a Farley Score boundary as part of my code reviewer checklist to enforce quality standards.

This kind of cross-cutting checkers (like a Design System compliance checker or a Security Compliance Checker) are useful to keep your agents from deviating from the general architecture, look and feel, and keep it maintainable in the long run.

[↗ Test Design Reviewer](#)

-  Another worth trying is Jordan Coin's **Coding Skills** which attempts to evaluate the conformance to KISS, DRY, SOLID, YAGNI, Separation of Concerns, Law of Demeter, Boy Scout Rule, Convention over Configuration, and so forth.

[↗ Jordan Coin's Coding Skills](#)

Meta Agents

Agentic systems produce a lot of logs, which turn out to be rather laborious to parse through. So it's handy to have special 'meta' agents to help you in these tasks. I'm still sitting on the fence to what degree this should be automated, as in the end, we're essentially using the AI to fix/guard AI, and the risk of polluting your system more than help is there, and the signal-to-noise ratio might not always be very good.

Did you know Claude was named after Claude 'Information Theory' Shannon? Wonder what he'd think about all this. Would he think that 'prompt is a lossy encoding of intent?'. Or, would he indeed 'enjoy analyzing the error patterns of LLM-generated code statistically' as much as we do, and the model named after him speculated (yeah I asked about this).

Anyway, **Meta agents** operate across the pipeline rather than within a specific stage. Here's some I've introduced in our setup

Agent	What It Does
Best Practices	Verifies agent specification files
Learning Loop	Captures operational insights and patterns from agent runs to inform future improvements
Document Librarian	Ensures project documentation stays in sync with code changes and possible patterns identified by the Learning Loop

This "Meta" category might turn out to be a very important part of any agentic factory. Given that tokens are expensive and good models are sometimes rather slow, increasing the chances of 'one-shotting' any task is a good investment.

Typical problems I'm trying to solve with these agents range from **context overflows** to **tool call fights** to **token and time eaters**.

Three most common 'antipatterns', or 'problem patterns' I have observed so far include:

- **Context overflow:** When an agent's input exceeds the model's context window, leading to truncated information and poor performance, or possible retries to run the same task again and again until it succeeds. Obviously, this is not a good solution.
- **Tool call fights:** When an agent gets a tool call wrong but instead of giving up, ends up making the same call over and over again.
- **Token and time eaters:** When the task was just about adding some simple thing onto, say, a page, the agent ends up reading half of your 200,000-line codebase for examples and the run takes 30 minutes, ultimately failing.

The list is, naturally, by no means conclusive. It should be noted that all of them are more or less about the deficiencies of the LLMs: randomness, limited memory (context window), and their agent harnesses (such as CC), but also about the complexity of the task and the process as a whole.

Detecting this kind of errors just by looking at outputs is a job in itself. So, having an agent to do it for you is a no-brainer. Who knows, perhaps one day we might have a fully self-healing system of AI agents detecting their own poor performance, and **self-learning** capabilities which adjust the instructions, recipes, tooling, and so forth as the learnings accumulate automatically.

Agent recipes

Each agent is defined in a Markdown specification file. The 'frontmatter', which is sadly not yet really standardized but a tool-specific thing, contains the agent's name, description, and the tools it can use, and depending on the tool, perhaps the model to call, handoff points and so on.

The body of the file contains detailed instructions on how to perform the tasks for that phase, along with constraints, references, and criteria for success and failure.

These specifications are the contracts that bind agent behavior.

My idea of an agent blueprint is organized as follows. Additional sections are used as needed for the type-specific things, but roughly these should exist.

Purpose	What this agent does and doesn't do. The intent.
Allowed actions	The specific operations this agent may perform (basically tools)
Entry criteria	What must be true before the agent starts work
Exit criteria	What must be true before the agent can advance the gate
Error criteria	What conditions would cause the agent to fail and require your intervention
Workflow	How to execute the tasks given
Success criteria	What are the success criteria for this agent's work, and what should be checked before advancing the gate
Constraints	Do's and don'ts
References	Links to compressed index documentation relevant to this agent's work.
Feedback and learning	How to discover and persist lessons learned from this agent's execution for future improvement (highly experimental still)

The primary user interface is VS Code's chat panel via the **Orchestrator**, or manually by selecting individual agents. The agent specification file, or the 'recipe', acts as the system prompt for the agent and defines the guardrails that keep it aligned with the pipeline's rules and expectations along with task-specific dos and don'ts.



The **Best Practices agent** is my development environment gatekeeper. It reviews specification files against patterns we learned work well: clear constraints, explicit skill requirements, measurable exit criteria, and so forth. I'd advise you have something like that too, and run it periodically or when changing the agents.

Gate-based workflow

Tasks progress through my agentic software factory through five sequential gates. These gates are guarded by a CLI tool called `task.sh`. This is the state machine I've discussed earlier. In its simplest form, the gates are:

Gate Transition	Prerequisites
Planning to Engineering	Plan validated and approved by you
Engineering to Testing	All implementation tasks completed, build passes, documentation checked
Testing to Review	All test tasks completed, test quality score meets threshold (Farley Index ≥ 6.0), documentation checked
Review to Complete	Reviewer approves; on rejection, you reset the gate back to engineering

The critical design decision was that the **gates are enforced by the CLI, not by agent self-discipline**.

Agents must call `task.sh validate-gate` before starting work and can only call `task.sh set-gate` to advance when all exit conditions are met. To be honest, which should come as no surprise to anybody working with LLMs, sometimes they are still a bit stubborn and selective about this though.

The tool validates state transitions and rejects invalid ones. This prevents agents from running out of order, skipping steps, or the hallmark of LLM arrogance, **declaring victory prematurely**. It also prevents the agents from modifying the state files directly, which could only end in a tragedy.



Don't ask probabilistic systems to enforce deterministic rules. They aren't designed for that. Some agent harnesses have capabilities like hooks to enforce actions during agent lifespan. You might find them to be useful for this kind of enforcement, too.

[↗ GitHub Copilot Hooks](#)

Task tracking and status management

The `task.sh` CLI serves as the single interface for all status operations.

It manages a three-level hierarchy:

- **Features:** top-level work items, each containing one or more stories
- **Stories:** individual user stories, each with a gate (current phase) and a task list
- **Tasks:** atomic work items with states: `pending` → `in-progress` → `completed` / `failed`

State is persisted as JSON files under the working folder. To keep them on track, **agents are prohibited from reading or editing these files directly**. (This is what the `.lignore` is for.) So all state and query operations go through the CLI, which validates state transitions and maintains data integrity.

✓ In order for your agent factory to work, you need to have a **solid tool to maintain the production process**. The task-tracking is our shift foreman and factory manager, which allows us 1) to work in separate branches by different people, 2) to have a traceable trail on why something was done and perhaps adjust that later without losing context, and 3) makes sure guardrails are actually followed. *Task-tracking skill* is used by all agents to record, update, and query the status and planning materials instead of the wild collection of CAPITALIZED.md files scattered around.

Key CLI commands include `list-features`, `progress`, `init-story`, `set-gate`, `validate-gate`, `start-task`, `complete-task`, `add-test-task`, `next-task`, and `next-action`. The last one deserves special mention: `next-action` returns a machine-readable JSON recommendation for the Orchestrator, telling it which agent should run next and what state the pipeline is in. This is how the **Orchestrator** stays stateless (or that it can always resume where it left off).

The skills library

Skills are reusable knowledge modules that agents invoke for specialized tasks. Each skill has a `SKILL.md` file defining when and how to use it.

Make a `SKILL.md` as an frontend for all CLI and other tools that execute code, queries, or other operations. They might also be additional 'recipes' for agents not always needed. The core promise and difference between skills and tools is the load-on-demand: only the frontmatter of the skills is loaded up front, and rest only when they are indeed invoked.

Here's what I've found useful.

Category	Skills
Code quality	Architectural code review, code smell detector, code metrics
Testing	Farley properties and scoring, signal detection patterns, test design reviewer
Reverse Engineer	A reverse engineer to produce design document from existing app
API development	API types generator (executes live backend calls to produce TypeScript types)
Project management	Task tracking, Azure DevOps use case analyzer, plan quality checker
Meta / tooling	Skill creator, agent MD refactor, browser automation

I used several skills available online as basis for many of the above.

I even found an agent compliance checker, which helps me with the tuning of this little factory. Go ahead and check the links below and don't reinvent the wheel.

[↗ Skills Library](#)

[↗ Awesome GitHub Copilot](#)

A word of caution though: **Don't copy & paste agents and skills directly and expect them to work well.** Read what they do and whether they align with the process you want to build. Don't be afraid to modify them to suit your context: most of them are just text.

Another tip regarding skills I've learned the hard way: Use the *exact name* when you refer to a skill in your agent. Like add 'load this skill from .skills/this.md' in your agent's recipe to make absolutely sure it will. (oh well)

Some have even adopted a 'debug printout'-style to keep track of the skill usage to be printed every time a skill is used (they are not as easy to spot from agent logs as the tool calls).

Like: 'Print INVOKED every time you've been loaded'

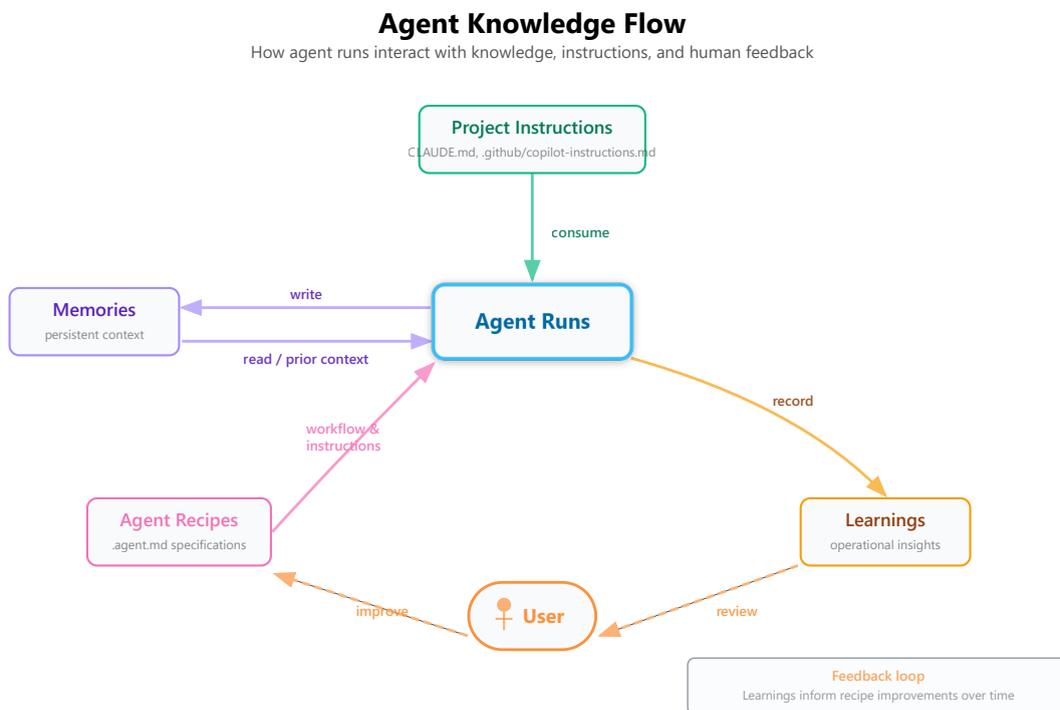


One of the major lessons when working with agents, tools, subagents, MCP servers, and whatnot has been that for an old programmer it's hard to adjust to non-deterministic systems. No matter how exact your steps, skill references, and tool names, sometimes the damned things just don't use them. So test, and use the self-learning techniques to tune your prompts, recipes, and all to increase the chances your agents stick to their designated paths and tool sets.

Memories and learnings

Memories are automatic, contextual data points that agents can access to avoid known pitfalls. For example, if the Engineering Agent encounters a build failure due to a common misconfiguration, it can invoke the memory system tool to check if this issue has been encountered before and what the resolution was. This prevents agents from repeating the same mistakes and allows them to learn from past experience without human intervention. This type of knowledge is perhaps something specific to a certain user, and often not worth sharing with the team, so I would suggest keeping it local and not sharing it via git. I've often found myself disabling the tools completely.

Learnings are a collection of analyses of an agentic task. For instance, they could be the 'error logs' of multiple retries, things that break, detected patterns, and so forth. They should not be automatically surfaced like memories, and probably need a human eye before deciding the correct action, but serve as a backlog of things to fix, refine, or adjust in the upcoming iterations.



Agent knowledge flow: how runs, memories, learnings, and recipes connect through a human feedback loop

The figure above illustrates the flow of knowledge between agents, memories, learnings, and you. Learnings and memories are separate but complementary: memories are for quick retrieval of specific solutions, while learnings provide a broader context for continuous improvement. Some kind of learning/feedback loop is essential for creating a self-improving agentic system that can adapt and evolve over time.



There's a tool called memory as standard in the VS Code toolset. The basic idea is to store "memories" about your work and dynamically discover them later. Frankly I haven't found these very useful, and as it is with this scene, think they are somewhat overlapping with other concepts. I would suggest disabling them at first and only enabling them if you find a specific use case, and even then be careful about sharing them via git. Keep it simple, folks. For a graph-based memory engine, which really works but I couldn't figure out a good use for, the link is right below.

[↗ Knowledge Graph Memory Server](#)

To iterate the agent recipes, and potentially common project instructions, I've incorporated a rudimentary **learning** system for the agent recipes. Tbh that's not really yet battle-tested, but in principle, it records operational insights under a `learnings/` directory, organized by phase (planning, engineering, testing, review). These capture implementation pitfalls, flaky test causes, common review findings, and reusable patterns (such as repeating code). Unlike memories (which are automatically surfaced), learnings are more like a project wiki that agents and humans can consult, and as information something more fine-grained and perhaps not persistent. They should be cleared out periodically and moved to project documentation when they are deemed valuable.

An example configuration in an agent recipe looks like this:

```
## Learning & memory
```

```
Before starting implementation, check learnings from previous stories:
```

- Read ``learnings/engineering/common-pitfalls.md`` for recurring implementation issues
- Read ``learnings/engineering/implementation-patterns.md`` for reusable patterns discovered

```
After completing all tasks for a story, persist new learnings:
```

- Document candidates for reusable patterns or code identified during implementation
- Record pitfalls encountered and their resolutions, such as misuse of hooks or API misunderstandin
- Store new insights in ``learnings/engineering/`` for future reference in status 'draft'



Come to think of it, this should be a SKILL, not part of the agent recipe. Here's continuous improvement in action!

Branching and PR workflow

The branching model I use is straightforward and strict:

- A feature branch is mandatory before any code or test changes
- Naming convention: `feature/<feature-id>-<story-id>-<short-title>`
- Work never happens on `main`
- After all stories in a feature reach the `complete` gate, the branch is rebased on `main`, the full test suite runs, and a PR is created (squash merge)

This is standard Git workflow, but enforced as a pipeline constraint rather than a team convention. The Engineering Agent cannot write code without first being on a feature branch; the pipeline will not advance to the PR stage without all stories completing their gates.

Human checkpoints and control

I still prefer that a living creature acts as an **approver and circuit breaker** at every phase boundary, at least for anything more complex than adding a button to a form which triggers an API call.

As a principle, at the end of the workflow, each agent should stop and present their output for review before declaring their work done. In my setup, you can pick up the Orchestrator and let it run for a while, and even instruct it to proceed automatically, but in my experience in the ramp-up phase it's better to keep tabs on it, and 'loosen the leash' only when you are sure the agents are doing what they are supposed to do. Trust, but verify, as they say.

In short, our current system expects you to:

Trigger workflow execution	by invoking the Orchestrator and giving the task to work on (e.g., "Begin next story in Feature 1")
Approve gate transitions	to let work continue
Reject and reset gates when needed	(e.g., from review back to engineering)
Provide guidance	when agents encounter build failures or ambiguous requirements
Defer non-critical suggestions	(e.g., documentation updates) without blocking progress

A sample run

Next, I'll describe a realistic run through the pipeline. To be 100% transparent, I didn't take a real-world example here, but you need to take my word for it that this is exactly how it looks when I run the system.

So, let's say I have (well I do) a feature-level product backlog that looks like this:

```
Product Feature 1 (a generic area of functionality, e.g. "User Management")
- User story 23 (a high-level user story with acceptance criteria, e.g. "Login")
- User story 34 (a high-level user story with acceptance criteria, e.g. "Logout")
```

For each of the **product backlog** features, we have prepared a similar project backlog:

```
Phase 1: Feature 1, Feature 2, Feature 3, ...
Phase 2: Feature 4, Feature 5, Feature 6, ...
```

All of this is still entirely in our ticket system.

Let's take a walk-through of how the system would handle the implementation of "Feature 1" with its two user stories, "Login" and "Logout".

Upstream: Backlog refinement and story generation

First we'll get the big picture and correct high-level scope ready.

► Example

```
Pick the feature from the backlog. This could be something like 'User Management', 'Order Management', 'Inventory Management', etc.
```

```
Review the feature and refine it.
```

```
Verify other reference materials and review possible changes in our change request log, stakeholder feedback, and other relevant project documentation.
```

Enhance as necessary.

Once you've refined the feature, it's time to feed it into the pipeline. The system is designed to follow the feature/story/task-division, so e.g. a feature with three user stories from 'Project Bootstrapper' will go through the rest of the pipeline three times. The Bootstrapper is kind of our 'sprint planning' or 'backlog grooming', where we fill our **project backlog** ready for the actual implementation work.

▶ Example

'Downstream', once/feature:

Select the 'Project Bootstrapper'

- › **You:** Bootstrap the feature 1
- ▶ **Bootstrapper:** Working ...
- ▶ **Bootstrapper:** Review the candidates to include in the project backlog.
- ▶ **Bootstrapper:** Story 1: Title
- ▶ **Bootstrapper:** Story 2: Title

Review and edit the list, remove irrelevant ones, add missing ones.

- › **You:** List is complete. Continue.
- ▶ **Bootstrapper:** Working ... (Read user story, extract the flow and acceptance criteria)
- ▶ **Bootstrapper:** *(apply PRD template to generate the user story drafts)*
- ▶ **Bootstrapper:** *(create tracking entries to the task-tracking system.)*
- ▶ **Bootstrapper:** [Story 1] User story draft with acceptance criteria and tracking entry, created along with DoD
- ▶ **Bootstrapper:** [Story 2] ...

Review the tasks

- › **You:** OK, let's add the automatic logout story, which is missing.
- ▶ **Bootstrapper:** Working... (creates tracking entries for each story with the task-tracking tool)
- ▶ **Bootstrapper:** Here's the stories:
- ▶ **Bootstrapper:** [Story 1] User story draft with acceptance criteria and tracking entry, DoD

- ▶ **Bootstrapper:** [Story 2] ...
- ▶ **Bootstrapper:** [Story 3] 'Automatic logout'

Review once more

- › **You:** OK! Let's move the feature to the project backlog and start planning.
- ▶ **Bootstrapper:** Working... (moves the feature to the 'Planning gate' with task-tracking tool.)
- ▶ **Bootstrapper:** Working... (creates user stories to the ticket system team backlog)

Now our ticket system backlog looks like this:

Feature 1

- Story 1 Login + DoD + Acceptance criteria
- Story 2 Logout + DoD + Acceptance criteria
- Story 3 Automatic logout + DoD + Acceptance criteria

i Time spent: ~1 hour

Now we have most of the product-side legwork done!

Downstream: The pipeline in action

Planning and Design

So, now we have a user story in the 'Planning gate' and it's time to start the actual implementation work. The story in the ticket system gets assigned to somebody like me.

▶ Example

- **1. Select the 'Orchestrator agent'
- › **You:** Plan next story in Feature 1
- ◆ **Orchestrator:** Working ... (finds the story 1 on the 'Planning' gate)
- ◆ **Orchestrator:** (*spawns the Planner agent*)
- ◆ **Planner:** Do you have existing code and a design for a new one?

2. Add context to design guidelines

- › **You:** Use Login.vb from the legacy codebase as reference. No new design, follow our style guide.
- ◇ **Planner:** *(reads the user story, acceptance criteria, and reference materials)*
- ◇ **Planner:** *(realizes this is a new screen, so spawn a Screen Bootstrapper subagent)*
- ▶ **Bootstrapper:** Working ... *(reads ui.md from our docs)*
- ▶ **Bootstrapper:** *(analyzes legacy codebase for relevant patterns, constraints, and strings)*
- ▶ **Bootstrapper:** *(splits the screen into components, identifies the API calls and data models)*
- ▶ **Bootstrapper:** Here's the design document (*)
- ▶ **Bootstrapper:** Here's the task list for the engineering phase:
- ▶ **Bootstrapper:** Task 1: Implement the login form component
- ▶ **Bootstrapper:** Task 2: Implement the API integration for authentication
- ▶ **Bootstrapper:** Task 3: Implement the session management logic
- ▶ **Bootstrapper:** Task 4: some totally irrelevant task

3. Review the plan and tasks, refine as needed

- › **You:** Remove task 4, which is not relevant. Add a task to implement the "Remember me" checkbox.
 - ◇ **Planner:** Working... *(updates the plan and task list accordingly)*
- and so forth.

(*) [A Markdown file with the visual layout of the new login screen, along with notes on which components to use and where to find relevant code snippets in the legacy codebase]

i Time spent: 30 minutes

@ Here's what an actual plan artifact looks like. This is the GSD output for the roguelike's first implementation phase, library basement generation:

```
phase: 01-library-rooms
plan: 01
type: execute
wave: 1
depends_on: []
files_modified:
  - js/map.js
autonomous: true

must_haves:
  truths:
    - "Floor 1 generates as library basement (not generic dungeon)"
    - "Basement uses distinct wall and floor tiles with brown-gray palette"
    - "Basement rooms are medium-sized (between sewer cramped and dungeon open)"
    - "Light enemies spawn on floor 1 (mostly cultists, some ghouls, rare deep)"
    - "Special rooms (Temple, Cultist Shrine, Morgue) never appear on floor 1"
  artifacts:
    - path: "js/map.js"
      provides: "Library basement generation with tiles, routing, enemy picker"
      contains: "BASEMENT_WALL"
  key_links:
    - from: "MapGenerator.generate()"
      to: "MapGenerator.generateLibraryBasement()"
      via: "floorNum === 1 conditional"
    - from: "generateLibraryBasement()"
      to: "_pickBasementEnemy()"
      via: "enemy type selection during spawn"
```

Notice the `truths` field: these are the acceptance criteria expressed as testable assertions. The `key_links` field captures the expected call chain, which is exactly what the reviewer will check against. The `autonomous: true` flag means the engineering agent can proceed without human approval, because the plan is detailed enough. That's the payoff of the spec work.

Implementation, Testing, and Review

At this point, now that our *agentic factory* has been tuned to actually perform rather well if the 'specs are in order', we let the Orchestrator loose, keep an eye on it, and intervene only if necessary.

Factory... well let's call it a Workshop for now

► Example

- › **You:** Plan looks good. Implement the story 1 and stop at the PR stage.
- ◆ **Orchestrator:** Working... (spawns the Engineering agent)
- ⊗ **Engineering:** Checks and creates a feature-branch
- ⊗ **Engineering:** (*reads the user story, acceptance criteria, and plan*)
- ⊗ **Engineering:** (*starts task 1: implement the Login form component*)
- ⊗ **Engineering:** (*after each task, updates the status via task-tracking CLI*)
- ⊗ **Engineering:** (*after all tasks are completed, runs build validation*)
- ⊗ **Engineering:** (*corrects the error*)
- ⊗ **Engineering:** Working... (resumes, completes tasks, build passes, advances gate to testing)
- ◆ **Orchestrator:** Implementation passed... (spawns the Tester agent)
- ⊙ **Tester:** Working... (reads the user story, acceptance criteria, and plan)
- ⊙ **Tester:** (*designs end-to-end tests based on acceptance criteria*)
- ⊙ **Tester:** (*invokes the Test Design Reviewer subagent to score test quality*)
- ⊙ **Tester:** (*completes test implementation, runs tests*)
- ⊙ **Tester:** All tests passed, but the Farley Index is 5.5, which is below the threshold.
- ⊙ **Tester:** (*pauses and notifies you*)
 - 3. Read the test results and think what's missing and what to do about it. Decide to add one more test case.
- › **You:** Add a new test case for the "Remember me" checkbox.
- ⊙ **Tester:** Working... (creates a new test, resumes, updates tests, re-runs)
- ⊙ **Tester:** All tests passed, Farley Index is now 6.2, which meets the threshold. Advancing gate to review.
- ◆ **Orchestrator:** Testing phase passed... (spawns the Reviewer agent)
- ⊙ **Reviewer:** Working... (performs read-only code review against project standards, architecture rules, and quality metrics)
- ⊙ **Reviewer:** Code review passed with minor comments. Advancing gate to complete.
- ⊙ **Reviewer:** (*creates a PR with the implemented code and test changes*)

◆ **Orchestrator:** All done! The PR is ready for final review and merge.

Check the files actually changed from GIT and the proposed comment

> **You:** All good, push the PR.

◆ **Orchestrator:** Working... (merges the PR, updates the task tracking status, and moves on to the next story in the feature backlog)

Now there's a PR in the ticket system for my colleagues to review (and hopefully merge without too much back and forth). And I may pick the next story and rinse and repeat.

i Time spent: 30 minutes

I've omitted some steps done by the agents for brevity, but this is the general flow. The Orchestrator keeps the pipeline moving, the agents execute their specialized work, and you intervene at key checkpoints to guide and approve. For a rather small task, the actual runtime of this all is around an hour, which is not bad considering the amount of work being done.

Lessons learned

I've listed the biggest take-home lessons from building and running this system below. Yes, many other things went sideways at first, and also the 'agent harnesses' did develop and improve during the process, but these are the ones that I found to be most critical to get right early on.

Getting things really working is a big job. The documents hierarchy, indexing, project structures, skills, their new versions... And expect to revisit it weekly.

Be very explicit and brief when you give orders. 'Do task 1' etc.

Be very explicit in the instructions and use accurate names to refer to other agents, skills, and tools. Yes, including the casing. Remember that this game is not programming, but giving guidance.

Don't let agents edit state/status files directly. They'll screw 'em up, and all your instructions on how to do it.

Move as much as you can to CLI scripts and skills which just work the same every time.

Keep the agent tool list clean and short. The more 'memory', 'context7', 'ticket system x add ticket' etc. tools you add via MCP, the more likely the agents will get confused. And you won't need most of them.

Start a new session every chance you get, and make the agents spawn rather than working in the main session. Makes a big difference.

Tools update all the time. And break all the time. Freeze, stop responding, produce random errors.

THIS IS ACTUALLY FUN. The most exciting new technology I've worked with in almost 30 years.



Some may find both comfort and good enough solutions off the shelf, but my biggest takeaway is: **don't underestimate this effort.** Perhaps sticking to a more black-box system like Lovable spares you from (some of) this, but if you want or need to build your own, be prepared to spend quite a few hours to get it running well and reliably.

Future directions

I'm well aware that a good product that does everything I described above might be just around the corner and I might very well be **reinventing the wheel** here. Building this kind of system from the ground up, from first principles, is actually a lot of fun, and I have learned a lot in the process. This is part of my and hopefully many others' *engineering ethos*: understand how things work before building something on top of them.

So, the Factory 0.3 should have in no particular order:

Configurability and templates

A more user-friendly way to define agents, skills, and pipelines without having to write Markdown files and bash scripts. Maybe a visual editor or a higher-level DSL.

Context transfer and state management

A better way to manage context and state across agents and phases, perhaps with a shared knowledge graph or a more sophisticated memory system.

Proper feedback loops and learning

A more systematic way to capture learnings from each run and feed them back into the system for continuous improvement, perhaps with some kind of reinforcement learning or evolutionary algorithms. A standard logging format that captures as much detail as possible for future analysis could be set aside and reviewed regularly.

Faster iterations

Yes, many models are still quite slow, perhaps like $O(n)$ as a function of context size. Actually, the longer it takes the more likely it's gonna screw up something. I have a few ideas on how to speed this up, but look forward to OpenAI, Microsoft, and Anthropic doing that for me first.

Better 'WYSIWYG' pre-planning tool for UX

I do have a rudimentary single HTML export from 'screen bootstrapper', but having a 'StackBlitz-like' environment where you could iterate the looks and basic layout precisely before trying to integrate into your site/app might be a huge time-saver, and come to think of it, not even that hard to build.

Conclusion

In this chapter I've described the basic blueprint of my agentic software factory. Perhaps one day I'll release it as an open-source project, but for now, I hope this detailed walkthrough gives you a good sense of how to build and run your own system. No doubt you'll find similar (and probably better) setups online right away, but I think it's a useful exercise to see how one can be built from the ground up step-by-step, and what are the key design decisions and trade-offs involved.

! Building a good agentic framework adjusted to your context is a big job if you are doing something quite complex. Don't expect it to magically emerge by itself by purchasing CC or Copilot licenses for your team. Especially in a team setting, it will also require constant care. Consider automating the 'self diagnosis' and 'self correction' as soon as possible to detect failure/retry patterns, 'token eaters' and other anomalies.

13

The Specification Premium

What was forgotten during the Agile era

The acronym problem

If there's one thing IT people are creative at, it's coming up with TLAs and FLAs. The latest member of the family is SDD, Specification-Driven Development. Probably to align with Test Driven Development (**TDD**), Behaviour Driven Development (**BDD**) or Domain Driven Design (**DDD**). Or, how I referred to the Agile method, "Ticket Driven Development" (also **TDD**). Lots of D's.

Anyway, **SDD** is supposed to be the practice of writing detailed specifications for each feature before implementation. Funny that I thought something like this was supposed to happen all along, so I got on this ship already a bit annoyed.

While the acronym and the message are easy to make fun of, the idea is basically about giving your AI a good chance to find a good solution.

The winning formula

What if we could model the chance of success in a programming task with a formula? My hypothesis is that the probability of success (S) is a product of core factors divided by the task size (T). Let's also throw in a coefficient for model temperature and other factors that affect randomness.

$$S = \frac{A \times C \times P}{T} \times R$$

Where

Architectural quality (A): The degree to which the codebase has clear separation of concerns, well-defined modules, and consistent patterns. A well-architected codebase has a small solution space for

any given task, while a poorly architected codebase has an enormous solution space.

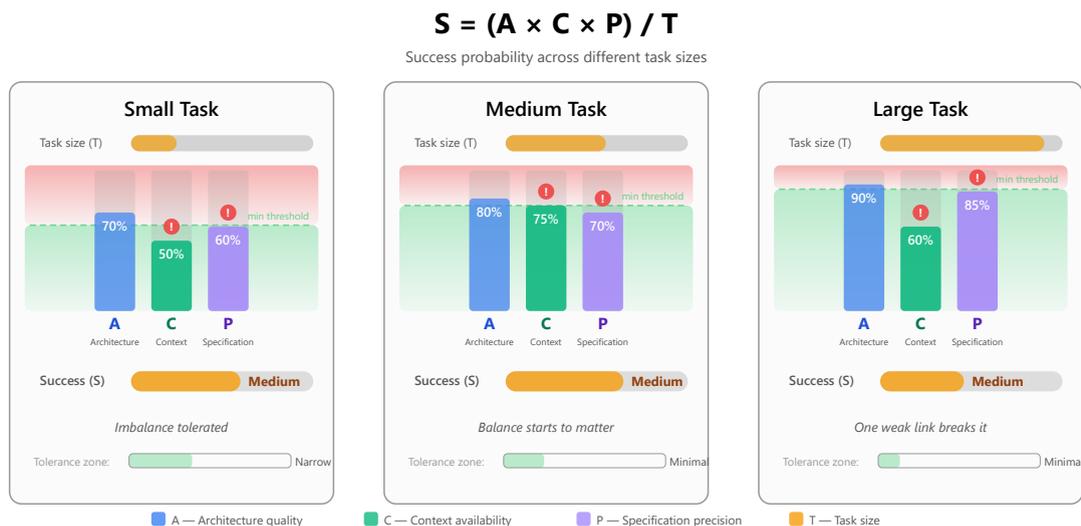
Context limits (C): The extent to which the agent can find and understand the relevant code and requirements within its context window. A codebase with clean architecture allows the agent to focus on a small subset of code, while a tangled codebase forces the agent to load vast amounts, increasing the likelihood of context overflow and dropped information.

Specification precision (P): The clarity and detail of the task specification. A precise specification with clear examples and constraints narrows the solution space. A vague specification leaves the agent to invent the approach, the "too creative zone."

LLM Fuzzy Factor (R): The inherent randomness in language model outputs, influenced by temperature settings and model architecture. Higher temperatures increase creativity but reduce reliability, while lower temperatures improve consistency at the cost of innovation.

Task size (T): The complexity of the task itself. Larger tasks have more potential solutions, more handoff points, and thus a lower success ratio.

Here's a simple take on what this looks like in practice. Imagine tasks of three different sizes: S, M, and L. It's still pretty easy to get things right even with lower values of A, C, and P for small tasks. But as the task size grows, you need to raise the bar: the bigger the feature you want to one-shot, the more attention to architecture, context, and specification is needed.



The winning formula: as task size grows, the tolerance for imbalance between Architecture, Context, and Specification shrinks dramatically.

Compare this with the "dumb zone" and "smart zone" analogy in "Concept Overload": The vaguer the architecture, boundaries, and context, and the bigger the chunk, the further your hardworking but essentially dumb assistant will wander from the actual solution.

The formula is anecdotal, based on a small sample size, not a controlled study, but it illustrates the point: specification is not just a nice-to-have, but a critical factor in the success of AI-assisted development.



Getting your specs in order is one central piece here, but not the only one. Strong specs can compensate for weakness elsewhere. If your specifications are rock-solid, you can take on bigger tasks at a time.

The specification premium

There was a good reason why writing software was painful in the old days. The lead time from planning to working software was measured in months due to intensive up-front paperwork, and the bar for changing course got high fast. Designs and specifications were written in stone, but already outdated once you started to work on them. Getting anything changed required a CR theatre and a minimum of two meetings and one month.

I thought the issue was never really about the specifications but about the non-incremental waterfall process. The general idea was sound, of course: by carefully planning things ahead, you would avoid costly mistakes and rewrites, and the odds of going the right way were higher. Often the problem was (and is) in the inflexibility and the long lead time from words to action.

*Spec-driven design in the old days was basically a **Hell of Writing Word Documents** to catch every detail of buttons, tables, columns, datatypes, and which UI control they appeared in. And the **use case diagrams**.*

So it turns out that **iterative development** is still a solid choice with the resurrection of detailed functional specifications. It does not matter if we call them "user stories", "PRDs", "ADRs" or "use cases": the important point is to let the AI know exactly what you want, under what circumstances, and what you don't want. Let's just not go back to 'a Word document which states which database column is which textbox'-thing.

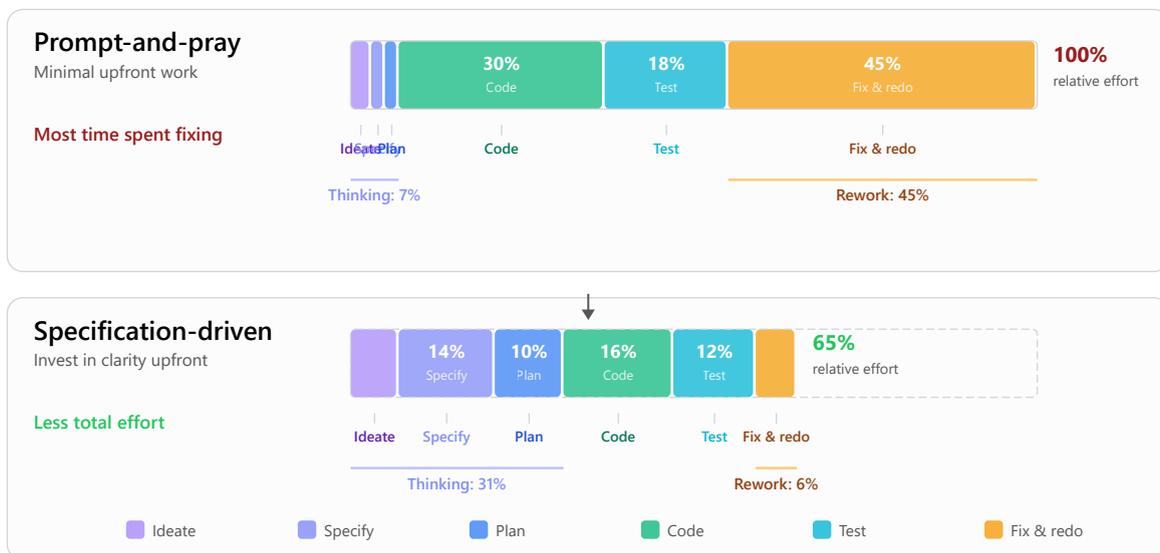
While writing documents is not entirely free, and often not the greatest fun, the cost of fixing and redoing is much higher. In my experience, having unbounded AI assistants write the code based on vague instructions just shifts the effort into the rework zone. Quite often the net hours spent would've been less doing it the old way.

Below, I've illustrated the **effort balance between writing good specs up front or fixing/redoing them later**. The numbers are for illustrative purposes, but probably not that unrealistic: when you take the 'left' part seriously, you can sometimes get those 5-10x speedups for the implementation phase.

In the end, it's your call if you want to jump directly to the coding part, and spend hours and hours later fixing that, or take that 10-15 minutes up front to write a good plan. Take your pick!

Where Does the Effort Go?

Same feature, two approaches — relative effort by phase



When you invest in ideation, specification, and planning, the fixing and redoing phase shrinks, and so does total effort.

Purely economically, I understand that promises about cheaper software don't fit well with the suggestion that work just moves to the specification phase and the same amount of effort is needed. On top of that, setting up a proper factory — documents, working agent chains, some self-learning capability — is easily a month's investment. And people like me suggest that you actually need new roles like the "AI Development Supervisor" to maintain the infrastructure: agent recipes, orchestration, supporting documents, and tools.

So let me be honest about where this pays off. For small projects with one or two developers, basic Plan and Agent modes with good project context is probably enough — the governance overhead isn't justified. For medium and large projects, the investment tends to pay for itself through reduced rework and repeatable delivery. As for token costs and licence fees: at current pricing, they are negligible compared to average developer hourly rates. Where that lands in two years remains to be seen.

Well, as we consultants say, there's no such thing as a free lunch, but some are considerably cheaper than others! The real cost-benefit question isn't "is governed delivery cheaper than ungoverned?" It's "is the rework you're doing now cheaper than the specification work you're skipping?"



Investing in **reasonably detailed** specifications and **planning in small steps** doesn't mean going back to the old days of Document-Driven Design. It's enough to write a "good enough" specification and let the AI help you refine it, find the relevant information, and synthesize it together before pressing run.

Structured artifacts, not vague descriptions

What is the optimal level of detail then? The answer depends on your team and context, so as usual, 'it depends'. The direction is clear, however: structured, detailed planning artifacts beat vague descriptions every time. They give agents unambiguous inputs, enable automated validation, and make the specification machine-readable without sacrificing human readability. For a practical calibration (how much spec effort different story types actually need), see [Chapter 16's](#) calibration table.

As usual, testing and automated testing were of secondary interest at first, and the test instructions for agents were rather high level. After a few weeks we had around 100 out of several hundred tests failing for common reasons: dependency on timing, lack of proper marking of UI elements, variable mocking strategies, you name it. Took me two days and quite a few tokens to fix and at least to some extent make them consistent. The lesson: practice runs with synthetic test material and careful review up front might reveal this kind of chaos before it spreads. If we'd specified exactly how to mock, how to name test cases, how to instrument the source code for testability, we'd have saved those two days and a lot of frustration.

Not all specification practices are equal. A study published by Thoughtworks examined specification-driven tools such as Amazon's Kiro, GitHub's spec-kit, and TESSL. They identified, by no less than **Martin Fowler** himself, three maturity levels that describe how tightly the specification is coupled to the code it describes.

Maturity Level	Approach	Spec Lifecycle
Spec-first	Write the specification before code	Created per task, may drift or be discarded once coding begins
Spec-anchored	Keep the spec alive through implementation	Persists alongside code, updated as requirements evolve
Spec-as-source	Spec is the single source of truth	Code is generated from spec; humans edit only the spec

➤ [Understanding Spec-Driven Development: Kiro, spec-kit, and TESSL](#) — Martin Fowler / Thoughtworks, 2025

Most tools operate at the **spec-first** level: they help you write a plan before coding, but the plan is essentially a detailed prompt that gets consumed and forgotten. A throwaway idea paper, basically.

The **spec-anchored** level is more persistent. The specification lives alongside the code and evolves with it. Specifications are versioned, referenced during implementation, and updated when requirements change.

The **spec-as-source** level is the most ambitious: the specification *becomes* the maintained artifact, and code is a generated derivative. This echoes earlier attempts at model-driven development (MDD), which never took off for business applications because it sat at an awkward abstraction level.

i If you're old enough to remember MDD, you might be thinking "we've seen this movie before." Fair. But the differences matter. MDD required formal models: UML, BPMN, DSLs, and specialized tooling like Rational Rose that cost more than your car. SDD specs are markdown files. The barrier to entry is a text editor and the ability to write sentences. MDD was also all-or-nothing: if the model was incomplete, the generator produced nothing. LLMs degrade gracefully: a vague spec produces mediocre output, a precise spec produces good output, but you're never stuck staring at a compilation error in your class diagram. That said, if this turns into "spec files that are as complex as the code they describe," we've learned nothing. Keep them simple or don't bother.

I'd personally go for a spec-as-source but refine-it-as-you-go approach. Have one source of truth, and don't let it be the code this time (despite it remaining that in the end).

Specifications force smaller steps

The step size principle from [Chapter 5](#) applies directly here: specification-driven design naturally enforces smaller steps because the act of writing a specification forces you to decompose work into units that can be clearly described. If you can't describe it clearly, it's too big.

And if you invest a bit in your tooling, like a '3rd Degree Interrogator Agent' that rips your spec apart and asks you questions about it until it's clear enough, you can **get to a good specification much faster than doing it manually**.

@ Every one of my game's ~200 features started as a specification. What the feature does, how it interacts with the player, what kind of color to use for the icon, where to render this or that. **The AI-driven interview**, where the agent asks clarifying questions before writing code, became my favorite part of the process. Not because it was efficient, but because it forced me to think about what I actually wanted before any code existed.

Context rot and how specification fights it

Qodo's 2025 survey of 609 developers discovered that while 78% reported productivity gains from AI coding tools, **65% said AI missed critical context during refactoring**. Refactoring is probably most dependent on understanding *how your code works*, seeking call graphs, dependencies, and matches.

'Your feature is now fully implemented and ready for production' typically means the total opposite.

To make it worse, these kinds of links are often masked behind complex conditional logic, a DI framework, a dynamic plugin, or DLL hell -- or **whatever evil thing the original developer was able to summon**. For example, let's take a three-level nested if block that calls something else, which calls something else, which calls something else, and so on. Perhaps by passing nicely renamed state variables and some !boolean logic down the path. **Good luck, AI.**

Naturally, the distinction between 'refactoring', 'fixing', and 'building new features' is not always clear, but the point stands. The study cited here found similar gaps in test generation (60%) and code review (58%).

Qodo's study also found that context gaps were cited more often than hallucinations as the root cause of poor AI-generated code. As you'd guess, **senior engineers were more likely to report these issues** and reported more frustration than juniors.

This is an example of 'context rot' in action.

When conversations or 'agent runs' grow longer, more files are read, tool calls are made (10 kB of JSON a pop), your CLAUDE.md is already 56 kB, half of it in ALL CAPS (**THIS IS IMPORTANT: [something the LLM keeps ignoring]**) -- it's pretty obvious your AI's context

window is full of noise and irrelevant information. Previous decisions, golden rules, and all fade from the context window, and there's just too much conflicting data inside.

Don't rot your context, man!

As a rule of thumb, the longer your prompt takes to process, the more likely it is not going to produce what you want. Unless it was a timeout, which the AI harness didn't bother to tell you about.

↗ [State of AI Code Quality in 2025](#) — Qodo, 2025

Nobody said the waterfall is back

Critics of SDD sometimes object that specification-driven design sounds like waterfall: big upfront design, rigid requirements, slow response to change. This mischaracterizes the approach.

My gut reaction to SDD was similar to Michael's in The Office after he found out Toby was back. 'NOOOOO'

Guess what, nobody said that is needed. You can **iterate as you like**, but need to go a bit further left than before, and that fantastic **AI guy is gonna write and cross-check most of it for you**. And neither Rational Rose, Visio, nor anything from Microsoft for that matter is going to be needed: nice, compact, and clean MD files will do, all revision-controlled and no 'Approved by: [insert name here] and Date: [insert date here]' needed on top of them either.

Yes, and you can refine them as you go.

How to not end up like Confluence

Every experienced developer has seen "living documentation" to not look very much alive. Confluence, SharePoint, whatever: the place where information goes to die. So why would specifications be any different this time? I think that's a fair question. Here's how I think about it.

First, have a clear single source of truth for how the system should behave from the user's or external actor's point of view. Link your implementation artifacts to it. After you've implemented a feature, do a quick sync pass: does the spec still describe what you actually built? If not, update it. Takes minutes, not hours. That's the difference between a living document and a fossil.

Second, don't try to document every little detail. Look and feel, pixel-level UX flows, individual field validations: these change constantly and are better expressed in code anyway. Your spec should capture *what the system does and why*, not every checkbox and tooltip. The detailed behavior? That belongs in tests. Tests are documentation that actually runs. When someone asks "what does this feature actually do?", the answer should be in a test case, not in a Word document from six months ago.

Third, ideally keep specs in the repo, version-controlled, next to the code, reviewed in the same pull request. If that's not possible (and I know it's not always possible), at least make sure your task artifacts point explicitly to the external source, even if that's Confluence. The worst case is when the spec exists somewhere but nothing in the codebase even references it. If your specs live in one system and your code lives in another with no links between them, they will drift apart. That's not a hypothesis, that's a law of nature.

Now here's what I think is actually different this time. When your agents use specs as input, specs become **load-bearing**. They're not decorative. When a spec goes stale, the next agent run produces visibly wrong output and somebody notices. A Confluence page that nobody reads can be wrong for years and nobody cares. A spec that your agents actually read and act on? That breaks loudly. And that changes the incentive structure entirely.

We've been here before, of course. UML diagrams that diverged from the code after the first sprint. Figma designs that stopped matching the UI after the third iteration. Architecture documents that described the system as it was six months ago. The failure mode is always the same: the model and the reality drift apart, and nobody has the time or motivation to keep them in sync. What might actually be different this time is that AI can work both sides. It can read the spec to generate code, and it can read the code to update the spec. The sync pass doesn't have to be a manual chore that everyone skips. An agent can compare what was specified with what was built and flag the gaps, or even propose the updates. We have a better fighting chance at maintaining a living description of our system than we ever did with Visio diagrams and Confluence pages. Whether that fighting chance turns into sustained practice remains to be seen.

And when spec and code diverge anyway (they will), that's a signal, not a sin. At 2 AM when production is on fire, nobody is going to update the spec before fixing the code. That's fine. Fix the

code, update the spec as part of the post-incident cleanup, same as you'd update tests. The spec is the *intended* source of truth, not an inviolable law. The alternative, pretending the spec is always correct, is how MDD died.

I'll be honest about one thing though: external-facing documentation is a separate problem, and I haven't really figured that out. Wikis, user guides, API docs, these tend to go stale regardless of how much you preach about keeping them current. Making them part of the regular agentic workflow is the right direction, but doing it well in a larger project remains a challenge. That said, the situation is still considerably better when you at least have structured specs and tests as a baseline. In most projects I've seen, the documentation amounted to chained JIRA user stories, an oddball `developer_readme.txt` nobody ever updated, and code with meager comments. The bar is not exactly high.

14

Developer Intervention Required

User checkpoints, testing as confidence, and the regeneration option

Shift left, verify right

Where should we engineers, testers, and other specialists in the software business shift our focus to make the most impact? One thing I've noticed, partly due to the immaturity of tools and processes, is that new roles will emerge. People need to rethink what their daily tasks will actually be. I see no return to handcrafting code line by line outside very specialist areas.

One dominant theme is the shift left, as already discussed in this book. More work needs to be done up front (left) and in the rear (right), letting the factory run the boring part in the middle. The explorative aspect of coding (or design) remains, but the tools are at a much higher level. In a way you can experiment and compare a hell of a lot faster than just a year or two ago.

We must also think about checkpoints, even hard stops where anthropogenic judgment is still required. What would these be?

I'd approach this puzzle by summoning the good ol' 5W+1H problem-framing technique:

Solution = What, Why, When, Who, Where, and How.

5W+1H assumes, appealing to those with an engineering mindset (*that's the tunnel vision a member of my family keeps talking about, right?*) that to solve something, these questions all need decent answers. We the engineers, and especially software engineers, have always been obsessed with the H (How) and somewhat less about the Ws. Certainly this remains a problem to this day.

Now, engineers with their TLAs and IDEs who are told to take on an advising role about the "How" might finally give attention to "What, Why, When, Who, and Where". Perhaps this could've been a good idea all along. Anyway, in more practical terms, this new RACI could look something like this:

5W+1H	Checkpoint	Validatoion	What It Catches
What & Why	Requirements approval	AI understood the requirement correctly, scope is bounded, problem is worth solving	Building wrong things before any code is generated
When & Who	Plan approval	Phasing makes sense, task breakdown is realistic, dependencies are identified	Planning failures, unrealistic scope, missing sequencing
What & How	Code review	Implementation matches the plan, tests cover acceptance criteria, quality standards met	Quality drift before code enters main branch
How	Architecture compliance	Solution follows architectural boundaries, conventions, and patterns	Architectural erosion, structural decisions that compound over time
How	Tooling evaluation	Current models, tools, and agent setup are still the best fit	Stale infrastructure, falling behind on capabilities

Obstacles to adoption

Getting people on board with these new job descriptions is going to be hard.

The specification discipline is new for many developers who, despite working in engineering, have been used to working like artists or craftspeople for quite some time. Luckily not in aeronautics or bridge building, but in software industry. We are used to making decisions on the fly and adjusting as we go.

Another challenge is that in practice **you'll end up verifying the same things all over again**, which are often wordy, repetitive, and not that fun.

Sooner or later you'll find yourself pressing "pass," "yes," "agreed," "do it" and move on without reading or checking anything. That's not always a problem as you can always revisit, adjust, and continue, but the risk of becoming too disconnected from the actual work is real.

5123 new/changed lines across 10 files to review. I've done that in less than a minute!

Hence:



The checkpoints should be kept small, intuitive, and focused on the most critical things.

The *AI fatigue* and kind of laziness will get you, too. It's just so tempting to throw smaller and smaller things to the fancy model just at your fingertips, and essentially to stop really thinking. Don't do that:

first it's not good for you, second it's a waste of resources, and third it will lead to more mistakes and rework.

A picture is worth a thousand words

To keep the review business viable, try to make it graphical.

In the example below, I've provided a simple ASCII graphic. This is what my AI UX Agent gave me to review. Not impressed? There were more details down to individual controls, but this was what I thought about most and found most valuable.

 Any intermediate planning artifacts for real persons must be visual, structured, and easy to review.

As rudimentary and boring as my example probably is, it represents a sanitized real-world example of a dashboard generated from a short description: "give me a page layout and implementation plan for this kind of data with this kind of hierarchy, with search and action list options."

With the rough idea, I could throw in (via project instructions) a basic design system, some generic code principles, and a similar page already implemented as an example.

Here's what I got:

```
+-----+
| PageHeader: "ProcessControl" |
| [Search: Seek with order...] [Date: DD.MM.YYYY] [Actions] |
+-----+
| ProcessList (Master View) Card |
| Tabs: [Active] [History] |
| +-----+ |
| | Id | Unit | Inf1 | Inf2 | Quality | Notes | Amount | ... | |
| |-----|-----|-----|-----|-----|-----|-----|-----| |
| | 12542 | A12 | X | | prem | ... | 10032k | [X] | |
| | 54344 | B05 | | Y | 2nd | ! | 1233 | [-] | |
| +-----+ |
| (flex: 0 0 40%, overflow: hidden) |
+-----+
| ProcessDetails (Detail View) Card outlined |
| Tabs: [Events] [KPIs] [BOM] |
| +-----+ |
| | [+ Add] | |
```

Date	Event	Notes	Start	End	..
1202	Bling	BP	14:30:15	14:32	1m

Compare this to reading hundreds of lines of text like this and trying to figure out what I'm going to get. Day in, day out, and then wait an hour or two to discover the thing was nothing I wanted. In case you wonder, no, **your visual drafts don't have to be the 80x25 ASCII art I so proudly showcase here** (mostly motivated by nostalgia I reckon). Go ahead and spin up an HTML version of it looking very close to the final product, and review it.

```
Drawers (rendered at page level, not in content area):
- RecordDrawer (right side)
- QualityDrawer (bottom, placement="bottom", size="large")
```

 Which of the above would you rather read? Use AI to generate summaries and graphical representations of the tasks, designs, and plans. Like a Gantt chart of the tasks, a diagram of the solution components and blast radius, a draft screenshot of the UI. All this is available at your fingertips.

While you can certainly still vibecode your way through all this, or "Lovablise" it, it might not give you anything solid to build on. The point of having somebody with a real brain in the loop is to keep the train on track, keeping things manageable and visual for the humans.

Your new role

You, in a governed loop, are not a programmer. You are an architect, a lead, a quality authority. The role shifts from *producing code* to *governing delivery*.

This is a genuine cultural shift. Most development organizations are structured around production: developers produce code, testers produce test cases, leads produce architecture documents. In a governed AI loop, the AI produces most of this. Your value is in the *decisions*: is this plan correct? Does this implementation meet our standards? Should this be shipped?

This requires different skills, different judgment, and different ways of measuring contribution. Organizations that try to run governed delivery with the old mental model—where "value" means

"lines of code written"—will find the framework frustrating. The value is in the governance, not the generation.

Organizations that embrace the shift often find their senior engineers are happier. The tedious parts of coding—boilerplate, routine transformations, repetitive patterns—are delegated to AI. The interesting parts—architecture, design decisions, quality judgment—become the focus of your attention.

Testing as the confidence layer



If AI output is probabilistic, testing converts probability into confidence.

Testing Layer	What It Catches	Why It Matters for AI Code
Unit tests	Logic errors, incorrect return values	Verifies the agent got the core behavior right
Contract tests	API mismatches, schema violations	Catches when agents invent or misread API contracts
E2E tests	User flow breakage, integration failures	Validates the full story works, not just individual pieces
Static analysis	Convention violations, architecture drift	Enforces patterns agents should follow but sometimes don't
Policy-as-code	Gate criteria violations, compliance gaps	Automates governance checks that otherwise require someone's judgment

In the governed pipeline, testing is a first-class stage with its own agent and gate. Tests must map to acceptance criteria before the gate advances. This is not an afterthought or a nice-to-have—it is the mechanism that makes probabilistic generation reliable enough for production.

The scope extends progressively: unit tests for logic, contract tests for APIs, E2E tests for user flows, static analysis for conventions and architecture rules, and eventually policy-as-code for gate criteria themselves.

This changes the economics of AI-generated code. When validation is automated and thorough, you can let AI generate aggressively and catch failures cheaply. The tradeoff shifts from "is this code perfect?" to "does this code pass all the checks?"—which is exactly the tradeoff that CI/CD pipelines already manage for code people write.

The more you can automate quality verification, the more safely you can delegate generation to AI agents.

The regeneration option

Here is perhaps the most counterintuitive benefit of a spec-anchored approach: if your specifications are good enough and your traceability chain is complete, you can—in theory—*regenerate the entire implementation*.

Think about what governed delivery produces for every story: a structured plan (architecture decisions, component mapping, scope boundaries), an ordered task breakdown with dependencies and deliverables, tests that verify acceptance criteria, and a complete traceability chain from story ID through branch, commits, and PR back to the backlog item.

If the AI models improve—and they will—you could take the same specifications and re-run the Engineering Agent with a better model. If your architecture changes, you could update the plan and regenerate. If a new framework emerges, the tasks could be re-scoped while the requirements stay stable. The specification becomes the durable asset; the code becomes (partially) disposable.

This reframes the relationship between specifications and code. In traditional development, the code *is* the valuable artifact—specifications are documentation that quickly drifts from reality. In a spec-anchored world, the specification is the valuable artifact—code is a (verifiable) derivation of it.

@ I never needed to regenerate any part of the game. But the specifications GSD produced for each feature are still there, the acceptance criteria still valid. I'd be curious how close a fresh run from those same specs would come to the original result. That's the real test of whether specifications are truly the durable artifact.

The regeneration option is best understood as a theoretical endgame that motivates investment in specification quality—a direction of travel rather than a current capability. Every improvement in specification precision, every addition to test coverage, moves you closer to this future. But teams should not plan on regeneration working reliably in the near term.

This is not fully realized today. Practical regeneration requires specifications precise enough that different agent runs produce compatible results, and test suites thorough enough to catch regressions. Current specifications are not that precise. Current test suites are not that thorough. The capability exists in theory but not yet in reliable practice.

Feedback loop design

The governance boundary is not just about control—it is also where learning happens. Each gate rejection is information: the plan was unclear, the implementation diverged from intent, the tests missed an edge case. Without systematic feedback loops, this information is lost.

Effective feedback loops answer:

What information from review flows into future specifications? When plan approval rejects a specification because the scope was wrong or the architecture would not work, that insight should inform how similar stories are specified in the future.

How does the team systematically learn from gate rejections? Patterns in rejections indicate systemic problems: unclear story descriptions, missing architectural context, inadequate test coverage. Tracking rejection reasons reveals where the process needs improvement.

How do specifications evolve based on implementation experience? When coding reveals that a planned approach will not work, the specification should be updated to reflect the new understanding, not just the code changed while the spec drifts.

The governed delivery artifacts make these feedback loops possible. Because specifications are structured and versioned, you can analyze them. Because the traceability chain is complete, you can connect rejections at later gates back to specification quality at earlier ones.

! The governance boundary is not just a control mechanism, but it is a measurement instrument for process improvement. Organizations that treat gate rejections as learning opportunities improve faster than those that treat them as annoyances.

15

Where the Framework Runs Out

Current state analysis

What I've been up to in my day job (which is not writing this stuff at all) has been to build a governed agentic development framework. I would've preferred to get most of it from ready-made tools, but after being in the scene I've found especially the governance side largely non-existent in the mainstream toolsets.

This book reflects what I learned while doing that and taking it to action. Small side projects I've used also in this book have covered the research side with more latitude of freedom, and I had fun doing it.

While the work I've done so far is actually doing a decent job, a lot of work remains. I realize keeping our agent setup up-to-date is a challenging task in itself. That's one of the reasons I think we will see new job openings for **Agent Orchestrators** or something like that, i.e. people responsible for taking care of your development infrastructure: agent recipes, orchestration, supporting documents and tools. As with the cloud, the result might be more complicated than what we used to have. (I've actually already seen some, and I'm slowly becoming one myself, too).

I'll go through some of my observations and open items in the table below, with selected points of view in more detail later in this chapter.

Problem	Current State	Why It's Hard	Potential Paths Forward
Circuit breakers	No good implementation yet	Current tooling doesn't support detection of wandering or looping	Monitoring agent outcomes, detecting divergence from requirements, automated halt triggers
Feedback loops	Review gates exist but closing the loop is manual	Gathering info on agentic failures and improving context engineering is still manual work	Structured failure analysis, automated feedback into context engineering
Learning from mistakes	Ad hoc at best	No systematic way to capture and learn from agentic failures	Coroner agent for post-mortems, pattern libraries, institutional memory

Problem	Current State	Why It's Hard	Potential Paths Forward
Pace of change	Framework is tool-agnostic but assumptions drift	Tools and categories shift faster than processes can track	Meta-practice: assigned owners, evaluation cadence, clear migration criteria
Human readiness	Many teams lack AI governance skills	Requires cultural shift from coding to governing agents	Training programs, skill development, architectural judgment emphasis
Measurement	Speed metrics tracked, direction metrics missing	Most orgs measure velocity, not requirement fitness or maintainability	Add flow metrics, track plan-stage catches, measure defect rates by governance
Security and regulatory gaps	Partial coverage via reviews and gates	AI-generated code introduces novel risks; legal situation still forming	Enhanced auditing, EU AI Act compliance tracking (August 2026 deadline), liability clarity
Directional velocity	Governance gates help catch directional errors	Organizations optimize for speed over direction	Culture shift needed toward valuing direction alongside speed
IPR and legal risks	Risk management, not resolution	Confidentiality, license contamination, and liability questions unresolved; law still catching up	Data residency requirements, explicit IP policies, human approval chains for defensibility
Sustainability	Not yet addressed	Energy consumption, skill degradation, overreliance risks	Monitor energy usage, maintain developer skill development, platform diversification

Circuit breakers

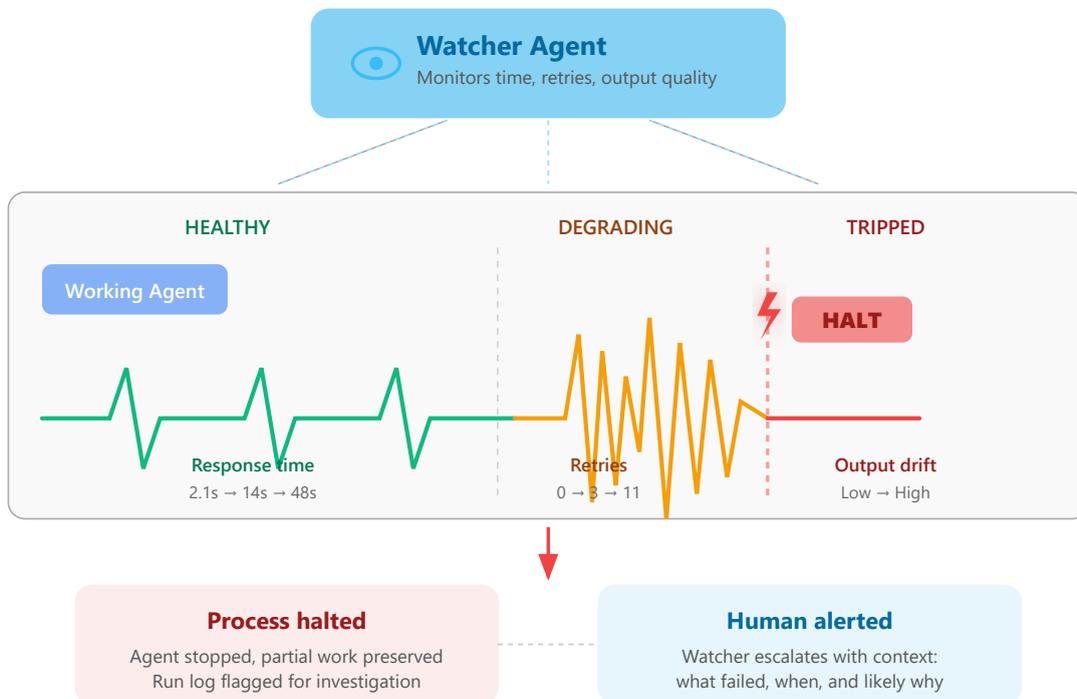
Circuit breakers are a critical safety mechanism for catching and halting negative feedback loops before they cause significant damage. In the context of governed agentic development, I haven't figured out a good way to implement them yet, and the current tooling does not support this very well either.

What I'm after here is the ability to detect wandering, going off the rails, or getting stuck in a loop. The idea is to have monitoring in place that tracks the outcomes of agentic development runs and identifies when things are going wrong. Such observations could be for example as simple as an agent repeatedly generating code that fails tests or other constraints, or even making repeated but failing tool calls.

When such patterns are detected, a circuit breaker would trigger, halting the process and alerting humans to intervene. From there you can investigate what went wrong: tool calls failing silently, documents not being followed, conflicting instructions, and so on.

Agent Circuit Breaker

A watcher agent monitors vital signs and trips the breaker when behavior degrades



A watcher agent monitors the working agent's vital signs: response time, retries or output drift. When some threshold is exceeded, it trips the circuit and halts the process or restarts it.

Given that our current setup is not yet the autonomous software factory running hundreds of agents in parallel in separate git worktrees, but still a human-triggered process, this hasn't been a priority-one item. It will become essential as we strive for more autonomy and scale.

@ In the game project, the agent actually worked pretty well within each task. The meandering and loops that plague larger scale solutions were rare and easy to spot when you're the only one watching. In a team setting that's not viable. The goal should be that agents perform their tasks independently and deviations are automatically detected.

Feedback loops

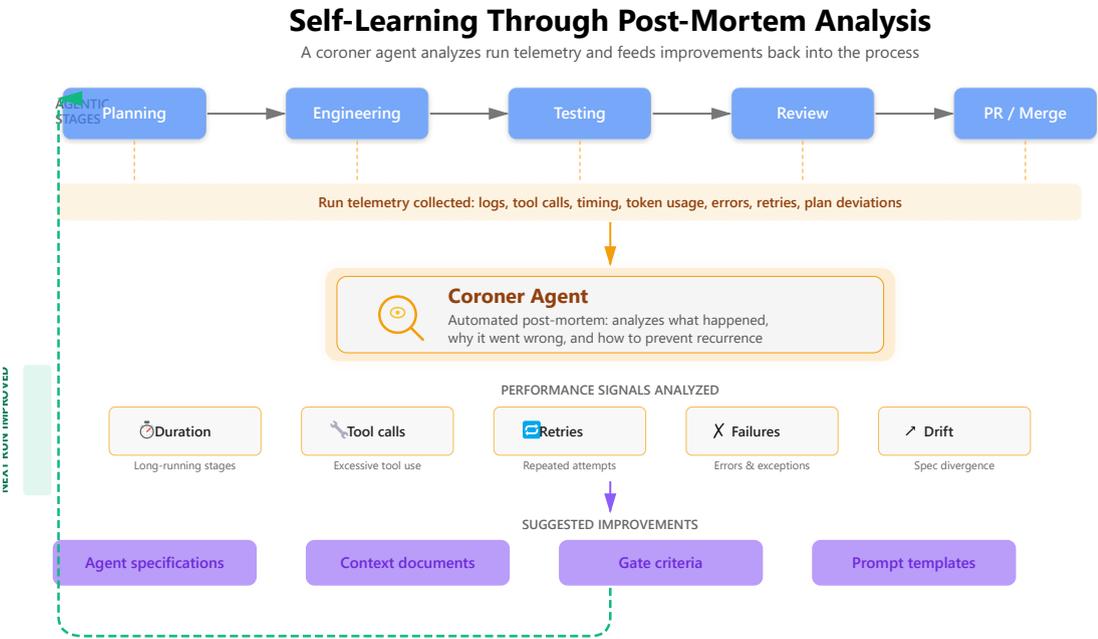
More about this later, but stemming from the circuit breaker topic, feedback loops are a broader challenge.

The current framework includes review gates and checklists that are designed to catch issues before they propagate, and help the developer correct course if needed.

The question remains: how do you gather information about agentic failures and improve the context engineering and agent framework to lessen the chances of repeating the same mistakes? Right now, detecting and analyzing failures is still pretty much down to somebody doing it manually. We've made some effort towards this, but it's still an open problem.

Learning from mistakes & reinforcement learning

A close relative to feedback loops is the idea of learning from mistakes. In traditional software development, when something goes wrong, developers analyze the failure, identify the root cause, and adjust their approach. If you are still on speaking terms with your colleagues, you might have a chat about it, too. In an agentic development framework, we need a systematic way to capture and learn from failures.



A coroner agent spawns after each run to analyze telemetry and feed improvements back into the process.

One way I've been thinking to semi-automate this is through a structured post-mortem process for significant failures. Imagine a coroner agent whose task is to analyze run logs, results, and changes, and suggest improvements to the context engineering, specification writing, or even the framework itself. This could create a feedback loop that helps the system learn from its mistakes and improve over time. We're still missing this piece.

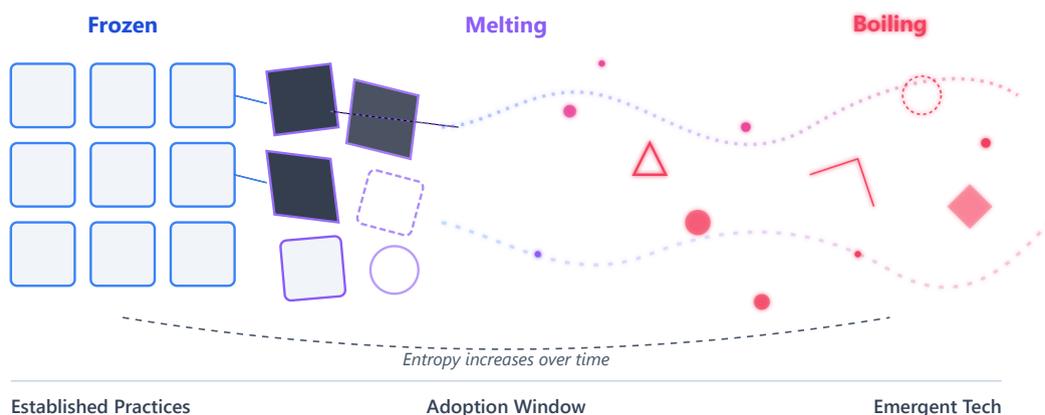
Dealing with the pace of change

Keeping up with the pace of change is one of the hardest practical challenges.

Even the basic tools we rely on keep adding new features and capabilities. This could be anything ranging, cross-tool compatibility changes such as new YAML frontmatter options, and other updates that can break the assumptions of your framework.

Imagine you've just gotten your agents, project docs, tools, and everything in order, ran a few successful runs, and then a piece of bad advice gets saved to a memory system and things deteriorate due to looping tool calls.

All of this is hard to debug because of the low transparency on what exactly is in the context. A hint: Github Copilot in VSCode offers very detailed logs that you can analyze to see what exactly was in the context and what the model was doing.



This is what it feels like.

What I've been trying to build is a *tool-agnostic framework* with patterns to manage planning, phasing, basic context engineering, agentic job coordination, and splitting work to specialists when needed. Together, these help (sometimes more, sometimes less) to get more done in less time with equal or better quality. I've wanted to keep people in the loop to prepare and review things. And despite our stack being very common, we still need people who can really code, debug and refactor.

If you're doing the same as I am, i.e. trying to bend what's possible right now into your settings, remember that just getting things up and running takes a lot of time, effort, and research. Keeping it running requires constantly revisiting what already worked. Compared to old-fashioned maintenance tasks such as

Dare I admit I DO have an agent named infrastructure-advisor, who despite the name not only advises but does. Don't worry, it does not have production access.

dealing with dependencies, build tools, and CI/CD pipelines, you'll have more to deal with. And all the previous things will still be there, too.

A good example of the field's immaturity: YAML frontmatter and skill/tool calling logic seem to change on every release. The frontmatter options are supposed to be "standards" but they're not. In practice, what works on one tool does not work the same way on another. So mixing different tools on the same project is a headache or at least a nuisance. As of early 2026, most common agentic tools are only beginning to read each other's configuration files, and their interoperability remains fragile.

Human readiness

Skeptics are there and they do have a point. The shift from "developer who writes code" to "delivery lead who governs agents" is cultural, not just procedural. Most developers were hired and trained to produce code. Governing AI-generated code requires different skills: architecture judgment, specification writing, quality evaluation, and the discipline to review what agents produce rather than rubber-stamping it. Chapters 5 and 6 explore this shift in depth.



The long-term question: will the ability to figure things out yourself become a lost art? If you haven't touched code directly for a long time, at what point have you lost the skill to do so?

The production feedback loop

Anyone ever built software that performed beautifully in development and testing but then failed in production? Yeah, me too. The feedback loop from production back to development is critical for catching issues that only surface in real-world usage. This time, it's even more important.

This is not an easy thing to solve. Even with real people in the loop, ensuring quality attributes are met (such as reliability, security, maintainability) has always been hard. These kind of *cross cutting concerns* by nature are determined by patterns and architecture, not as products of individual backlog items.

With agents in the loop, we need to capture production signals (errors, performance metrics, user feedback) and feed them back into the development process to inform future planning and context engineering. This requires instrumentation, monitoring, and a cultural shift in how teams think about the relationship between development and operations. This was the core tenet of the DevOps movement, and we might actually have a better chance to live by it than ever before.

Why this remains an open problem: the tools aren't there yet, and they are hard to integrate.

Measurement

Without flow metrics, you can't tell if the process adds value or ceremony. Most organizations measure speed (velocity, cycle time, deployment frequency) and almost none measure direction (requirement fitness, architectural coherence, long-term maintainability).

The framework needs measurement to justify its overhead: how much faster is story delivery with governance vs. without? How many plan-stage catches prevented expensive rework? What's the defect rate for governed vs. ungoverned stories? Without this data, adoption decisions are faith-based.

Security and regulatory gaps

It has been argued that AI-generated code is more insecure or buggy than what people write manually. It's not that simple.

Yes, AI is probably prone to use outdated patterns and sometimes insecure samples from its training data. It

can make stupid mistakes that a security-aware

developer wouldn't. The Fu et al. data from [Chapter 3](#) (security weaknesses in roughly a quarter of AI-generated snippets) is not reassuring. So the starting point is honest: AI-generated code needs at least as much security scrutiny as human-written code, and probably more.

*AI models are indeed very keen to create tests.
Sometimes too keen.*

But here's the other side. With AI, you can take your security auditing to a level that most teams never had the budget or the people for. You can build custom security scanning agents on top of existing tools like SAST, DAST, and dependency checkers. Don't ditch those, obviously, but you can layer AI on top to catch things the existing tools miss: business logic vulnerabilities, subtle auth flaws, misconfigured middleware. You can also use AI as a kind of fuzzy security tester, because it's naturally good at doing things slightly differently each run, which is exactly what you want when probing for edge cases. Very few organizations had the money or experts to do that kind of thorough security work before, let alone at this scale.

The net effect on security depends entirely on whether you actually use that capability or just let agents generate code and hope for the best. And regardless of what you do internally, anything going live on the internet should be audited by competent people, preferably a third party. That was true before AI and it's doubly true now.

On the regulatory side, AI introduces novel risks beyond just code quality: training data leakage, hallucinated dependencies, prompt injection vulnerabilities, and the tendency to generate plausible-

looking code that passes superficial review. The regulatory picture is still forming. The EU AI Act enforcement deadline in August 2026 will force clearer answers on liability when an agent introduces a vulnerability. The Cyber Resilience Act adds secure-by-design requirements and mandatory security updates for 5+ years. Whether your governance framework satisfies these requirements is a question you'll need legal help to answer, not just engineering judgment.

IPR and legal risks

Every prompt you send to a model provider carries your code, your specifications, and your business logic across a network to someone else's infrastructure. Most enterprise agreements include data processing terms and opt-outs from training on customer data, but the practical enforcement of those terms is trust-based. If your specifications contain trade secrets or proprietary algorithms, you're trusting the provider's isolation guarantees the same way you trust a cloud vendor.

✓ As a rule of thumb, **never store sensitive information of any kind on public cloud, and also don't send it to an LLM.** If you have that kind of thing, you'll need to run your own models.

Then there's the question of what comes back. Models trained on vast codebases can reproduce patterns, idioms, and occasionally near-verbatim snippets from their training data. Personally I think this might be more of an issue for creative work, such as novelists, than for engineers who've been de facto doing the same thing for years with Stack Overflow.

Anyway, the risk of introducing copyleft-licensed code into a proprietary codebase is real but nearly impossible to detect at scale. No reliable tooling exists yet to scan AI-generated output for license contamination the way we scan dependencies.

Liability is a sharper issue when you *run* an LLM as part of your product. Recent debates about how AI can be used for purposes like mass surveillance, or as tools that lead to lethal use of force, have put this topic again in the spotlight. I think it's different if you're generating user-facing content, analyzing data, etc. than when you *use* AI as a development tool. If you run an LLM that processes your data, inputs or outputs, or makes decisions during runtime — which, as I've explained to death in this book, WILL go sideways regularly — you have the responsibility for it, pretty much the same way you have the responsibility for any other software component in your stack.

Too bad Asimov's laws never made it to lawmakers' reading lists.

Taming the implications of having an LLM as part of your running system is not in the scope of this book, so we'll move on. For those of you who do that, best of luck.

Sustainability

Economics

How long will we enjoy nearly unlimited tokens per user at a few dozen euros per month? I'm not sure. Perhaps the next generation of LLM architectures will find vastly more efficient ways to produce those tokens. Perhaps specialized hardware will handle inference at a fraction of today's energy cost.

But the fact is that beyond Nvidia, not many are making money hosting LLMs. The biggest providers seem to be deep in the red. How long can they sustain that?

The longer-term risk is dependency. If we gradually lose the ability to create software without this technology, and the current consumer-friendly pricing goes up significantly, organizations that bet everything on AI-assisted delivery could find themselves locked in.

Environment and global issues

Since the early days of cloud computing, energy consumption of data centers has skyrocketed. The AI trend has been a step change in this steep upward curve. The numbers are huge, and it will not be just wind turbines powering the GPUs consuming your tokens.

Remember the "Energy Star" label on CRTs?

Then there are geopolitical risks. We've already built deep dependencies on technology and services from a small number of sources. There has always been a confidentiality risk, but now we might potentially face something more consequential. So, diversifying across providers and maintaining the ability to operate without any single one seems prudent. Fortunately, the LLM field has genuine global competition despite the two biggest players making the most headlines.

Education and skill development

Last but not least, echoed by voices from schools and academia: we have a generation of people in the pipeline who might never learn to write properly, solve equations, or summarize anything by themselves, let alone write code. It's a real concern.

The same applies to us adults. Why bother going deep into anything if you can LLM your way around it? We are by nature opportunistic and take the path of least resistance. Some of this is indeed valid and smart, but the long-term effects of outsourcing most of your thinking to language models could be significant.

Googling became a common verb but "chatgpting" didn't. I'm pretty sure Sam regrets the product name choice.

! Applying a framework described in this book requires people who understand systems deeply enough to govern them. If we stop developing that understanding, also governance becomes impossible.

16

The Beef and the Boilerplate

What actually works, what doesn't, and where to start

Before you begin

@ If you want to see governed AI development at the smallest possible scale, my game is the example. One developer, one agent, no infrastructure beyond a text editor and an API key. Everything in this chapter started there and grew outward.

In case you made it all the way here, you understand why accountability matters more than speed, why specifications narrow the solution space, and why gates enforce discipline.

Now let's talk about how to actually do it.

A typical starting point looks something like this. New project, new team, half the people have never used AI tools beyond ChatGPT, and the other half have used something different, individually, on a different stack. Everybody's eager. Licences are in, agents are running, and the first week produces an impressive amount of code. Looks great in the demo. Then week two hits and you notice: three different error handling patterns, two competing folder structures, tests that mock things in ways nobody agreed on, and a merge conflict rate that makes you question your life choices. The tools worked fine. The problem was that nobody had defined what "fine" means for this project. On top of that, the inherent randomness of outputs means that even with the same instructions, you get different results every time, and the sheer mass of generated code to verify is just too much too quickly. Getting to factory-level consistency takes weeks at minimum, often more.

What I'm going to cover next is:

- **The distinction** between work that requires your judgment and where agents perform
- **Calibration** for how much specification effort different work actually needs
- **Failure modes** that destroy value, presented as concrete anti-patterns

- A **phased approach** for building the capability incrementally
- **Metrics** that tell you whether the process is working
- A **checklist** you can use as a working template, starting next week.

A typical start setting you're likely to encounter is:



- ✓ The parent corporation has purchased the entry-level licences for some AI development tool
- ✓ The team has had some hours of training but no real practice beyond basic prompting and trials
- ✓ You might have some existing codebase and documentation, but no idea on how to bring them in. Typically they are not in good order.
- ✓ There's no tradition whatsoever to actually manage information or specify work in detail sufficient for agents to succeed
- ✓ Initial trials with default agents, such as planning and build agents have been unsatisfactory

The beef and the boilerplate

In order to figure out the correct level of abstraction for your specifications, you need to understand the fundamental distinction between two categories of work in AI-assisted delivery: the "beef" that requires real craft, and the "boilerplate" where agents earn their keep.

Next, I'll explain why this matters.

The beef: Work that requires your craft

The beef is the activities where your judgment is irreplaceable. This is where you say what to build.

For the beef to be actually digestible, let's say medium, we need to ensure that the following are in place:

Principle	Guidance
Ambiguity	AI can't infer the unstated assumptions or the implicit intent. Say what you mean. If you want a "user-friendly error message," specify what that message should say and when it should appear. If you want "good performance," specify the latency requirements and load conditions.

Principle	Guidance
Clarity	Keep to the *what* part. Focus on the outcome, not the implementation. "The list of items should be sorted by creation date by default" is clearer than "the API should use a sorted data structure." The former leaves room for the agent to choose the best implementation, the latter is implementation details.
Decomposition	Engineering 101: make a bigger problem solvable by breaking it into digestible steps with clean boundaries. This also allows for detecting dependencies and ordering tasks correctly. A story that will result in touching six services, three APIs, and requires a database migration is not a thing to be one-shotted by default, or at least this should be considered carefully; if it's more like adding a new field to UI then it might be OK. Depends. Same applies for features: "build a fully-featured CRM system" is not a single feature, but a sequence of features that build on each other, and you need to feed them in carefully planned order with guardrails and intermediate check points in between.
Architecture	Specify project-wide patterns, interfaces, constraints, and boundaries. Litter your documentation with examples. Your agents will eagerly generate thousands of lines of seemingly correct code but it's just different from anything else in the repo. Not only will this become a maintenance nightmare, but the future agent sessions will struggle in detecting the patterns if they haven't been applied in a systematic way in the existing codebase.



I've found it useful to refer to existing modules/code files/pages/components and whatnot as examples to compare to when planning a feature with our planning agent. This establishes a clear pattern for the agent to follow and reduces the chances of it inventing its own style that doesn't fit with the rest of the codebase. And if these discoveries are persisted in your plans already during planning, the engineering agent can follow them instead of redoing the codebase-wide greps and getting it wrong.

Acceptance criteria. The Pareto principle is very much alive in the agentic world as well. Much of the error often goes to the edge cases, however improbable they might be. Vagueness does not help here either: spend time, e.g., by using Gherkin format to define the paths that need to be covered, and ask AI to discover more. "It should work correctly" is not a criterion.

Review judgment is about knowing when output is correct versus when it merely compiles and passes tests.

We've all seen session ending like All code compiles and tests pass. And they don't.

Agents can produce code that is syntactically perfect, test-passing, and architecturally or functionally plain wrong. Only a person who understands the system's intent can catch these.

Other talking points you might have come across are that AI is **generating technical debt at scale** and introducing **subtle bugs** that are hard to detect. Both of these, among others, are valid concerns and to be taken seriously: it's a whole other ballgame to spend weeks refactoring things in the middle of the project or starting over, especially after you've already shipped something, with AI or not. Don't trust you can just "make the things right later". You really can't.

! If you delegate the beef, you don't save time. You move it to the review gate where it costs more, or worse, to production where it costs the most. Shift left, folks!

Skip the boilerplate: Where agents earn their keep

The boilerplate refers to things that should be rather straightforward. It consists of activities where agents reliably produce acceptable output. So 'boilerplate' basically refers to making the boring part automatic.

Examples of these kinds of activities include:

Task type	Guidance
Scaffolding	Project structure, boilerplate files, configuration. Given a convention file that says "we use this folder structure, these naming patterns, this test framework, in this kind of project"
Pattern application	Give an example page, control, or module, and ask the agent to follow that pattern for the new thing you need. Pattern can be anything ranging from naming, code file organization or an architectural pattern.
Test generation from specs	If you've done your homework in the 'beef' part well, this is relatively easy. You've already defined the acceptance criteria, so the agent's job is to translate those into test code. This is a perfect example of where agents can save time: they can generate thorough test suites that cover all the specified scenarios, including edge cases you might have missed.
Glue code	REST model to display model, data transformation, API integration. This is the "connect the dots" work that is tedious but straightforward. If you specify the inputs, outputs, and transformation logic, agents can reliably produce this code. Pass in a sample result, or better yet, the OpenAPI spec of your service, and perhaps a sample output, and let the AI do the translation work for you.
Documentation from code	Consider generating and maintaining the documentation per feature or per concern (like patterns) carefully. Something that's just basically slop and easily regenerated is not often worth it and will not be kept up-to-date, but as a feedback loop stage when you've for instance changed your API, data structures or introduced a new pattern or condition is a logical and necessary stage to review and refine.

! Unless carefully isolated, generated tests may be entirely rubbish and work on the wrong abstraction level, and in the worst case made to fit the code, not the requirements. Especially proper E2E tests need a black-box approach to test for outcomes, and often require explicit instrumentation (like data-id etc). AI is often overenthusiastic in generating way too many tests which make the test maintenance a nightmare.



This is something that might be a maintenance nightmare. You might have noticed your codebase being polluted by dozens of GUIDE.md IMPORTANT.md NOTES.md QUICKSTART.md REFERENCE.md in each folder. (Yes, Anthropic, it's you, hello!) In the end you'll have dozens of them with no idea if they are relevant or up-to-date. Be careful what you commit.

The guardrails for delegated work

An essential safety mechanism for agents to perform is to erect boundaries and hard guardrails.

On the top level, these fences are baked-in state tracking which requires certain tasks, states, and viewpoints to have been considered, and isolating the unit of work with branching and PRs from the already verified content.

Once you have a good agreement and tooling to support that, you should consider the feature-level guidance to keep things on track. Consider the following:

1. **Name the pattern.** Instead of vague "follow existing conventions," use "follow the 'Response Handling' pattern in `api.md` ." Yeah, however good your context engineering skills are, referring to tools and documents explicitly by **exact names** is sometimes the best way to go.
2. **Constrain the scope.** Some say that even more important than what needs to be done is the specification of what doesn't. So, state explicitly what's excluded: "don't add pagination", "don't change the database schema", "don't create new API endpoints beyond", "don't add new controls or pages" and so forth.
3. **Define the review criteria.** For example, "Check that code compiles" or "Verify that changes comply with Coding Conventions". This is something to be offloaded to your stage-specific review instructions and/or agent and should be performed in clean context/spawned subagent, preferably with a different model than the one who wrote the plan, document or code files waiting for commit.

How precise does the spec need to be?

The current generation of developers is not used to specifying work in detail and to be frank, it was not that much better pre-Agile either, or at least it was often different guys who did the plan.

So I've been thinking a lot about what's the suitable level of detail that's both realistic to get and detailed enough for the agents. Also a sanity check needs to be made: a quick bug fix, a small detail added or a small layout adjustment might not need anything on paper.

Chapter 13 made the case for *why* specification matters and introduced three maturity levels (spec-first, spec-anchored, spec-as-source). Here's the practical side: *how much* specification effort different work actually needs. Generally speaking, by the "The Calibration Principle" applied to agentic software engineering **Specification effort should be proportional to the cost of getting it wrong**. I'll postulate the following:

Story Type	Spec Precision	What to Include	Expected Error Classes	Review Focus
Bug fix (isolated, low risk)	Minimal: 2–3 sentences + test case	What's broken, expected behavior, reproduction steps	Wrong fix location, incomplete regression test	Does the fix match the symptom? Any side effects?
Small feature (single component)	Moderate: structured spec with acceptance criteria	Functional requirements, component boundaries, test scenarios	Missing edge cases, implicit scope assumptions	Does it do what was asked and nothing more?
Cross-cutting feature (multiple services)	Full: structured spec + architecture notes + task breakdown	API contracts, data flow, error handling, deployment order	Interface mismatches, ordering bugs, partial failures	Do the pieces fit together? Is the integration tested?
Architecture change	Full: ADR + multi-step plan + rollback strategy	Decision rationale, migration path, compatibility constraints, success criteria	Regression in existing behavior, missed dependencies, performance impact	Is the migration safe? Is rollback possible?

I encourage discussing these things with your team; it's nothing new and something that should be agreed even without AI. Also consider measuring these things like I suggest in **Knowing It Works** to get some data on the tasks which consistently come back for revision for bugs, missing features or architectural issues.

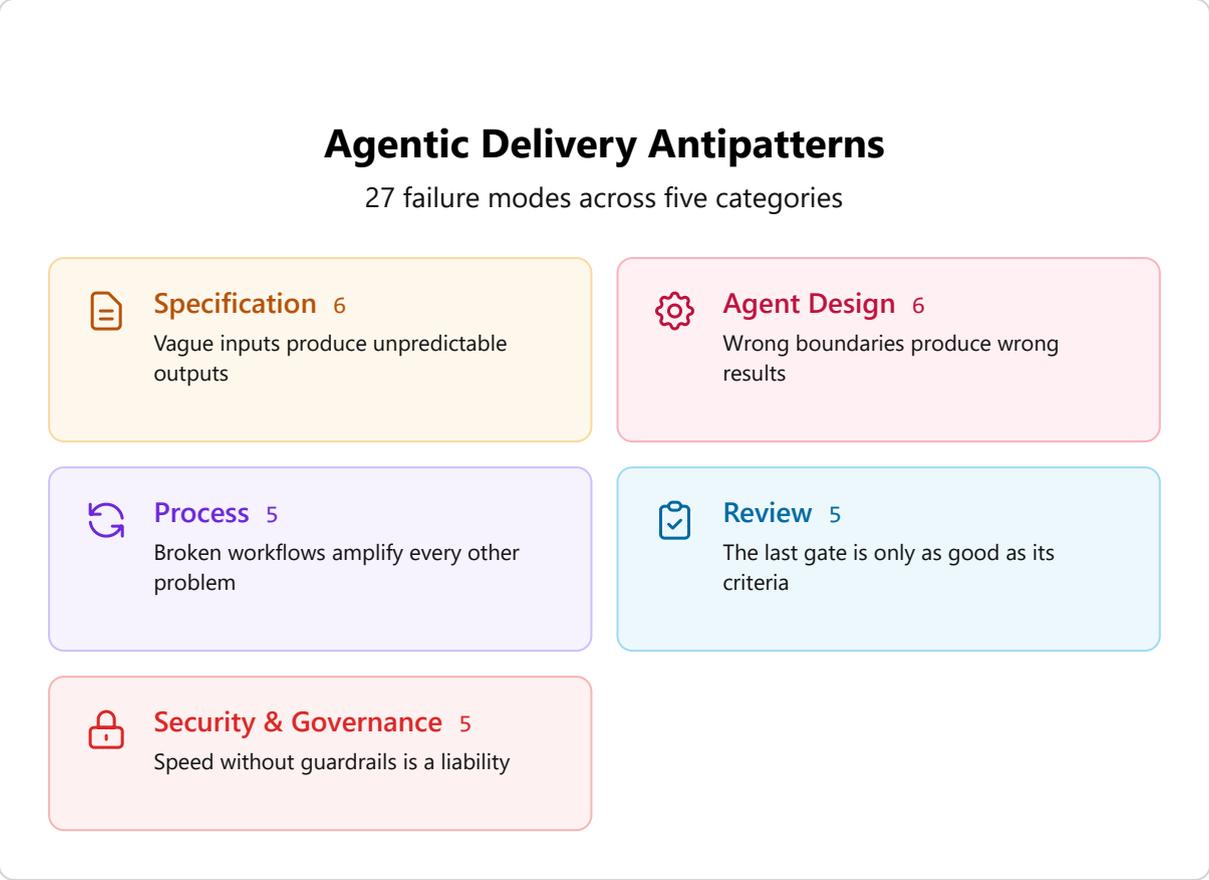


Look out for the ones that *look* small but aren't. Like a "simple UI change" that touches shared state, a "quick API update" that affects three consumers, a "minor refactor" that shifts module boundaries. Use AI (and your own judgment) to estimate the blast radius of changes and adjust your spec accordingly.

Antipatterns and bad, bad practices you should avoid

As I mentioned in the "focus on DONT's over DO's" principle in **Chapter 9**, it's often more instructive to look at what not to do than what to do.

This applies also on pattern level. I collected some well-known antipatterns below from the literature and some I've discovered in my work.



Agentic delivery antipatterns across six categories.

Many of the DONT's and DO's here are essentially an art of prompt engineering applied to the entire delivery process. So the change here is that we feed the prompts through the planning artifacts from files or similar instead of writing them ourselves. In the end, however, anything that you write or synthesize with AI is a prompt for the next step.

Specification anti-patterns

Here's what to say and what not when writing specifications for features or feature-independent general project documentation. Sure, you'll get the idea; the entire point is to be specific and explicit. Once you've set up your Software Factory and all the conventions and patterns properly in place, you can loosen the noose.

Don't	Do	What Goes Wrong
"It should handle errors gracefully"	"On 4xx, return error schema X. On 5xx, log to Y and return generic message. On timeout, retry once with backoff Z."	Agents often invent practices for error handling. Review catches inconsistency too late.

Don't	Do	What Goes Wrong
"Implement authentication"	"Add JWT session auth using middleware X, storing tokens in httpOnly cookies, with 24h expiry, following the pattern in auth-service.ts"	Multiple approaches across runs. Agent picks OAuth one time, sessions the next.
"Just use the existing pattern"	"Follow the repository pattern in user-repository.ts, including the error handling in lines 30–45 and the logging convention in lines 50–55"	"Existing pattern" means different things to different runs. Three implementations, three styles.
One story for a full feature	Break into: API endpoint, frontend component, integration test, migration. Each with its own spec and gate	Agent loses coherence on large tasks. Middle sections get less attention than beginning and end.
Omitting exclusions	"Scope: only the filter endpoint. Do NOT add sorting, pagination, or modify the existing list endpoint."	Agent adds features you didn't ask for. Creative completion is a feature of LLMs. You need to constrain it.

Agent design anti-patterns

Another group of anti-patterns I've encountered is about the agents themselves. Remember, agents are still LLM calls with a 'recipe' i.e. the intent, tasks the agent is supposed to do from your execution pipeline, and the context it should operate with (files, category, module). The last one could be already specified during planning, but the agent should be free enough to find more when needed.

To recap, remember that all these 'agents' are is: Agent = Recipe + Context + Tools

This is basically just an md file with thin YAML frontmatter. The 'standard' (note the hyphens) YAML is just a name, tool list, model to use followed by what's basically a prompt to the LLM. All agents should be general enough to perform any task your planning practice produces in the provided context.



As you learn more, go ahead and create specialists for different kind of tasks. Like a separate backend/frontend/integration/dba agent. Same bad practices will haunt them, too unless you pay attention.

Don't	Do	What Goes Wrong
Feed the entire codebase as context	Point agents at specific files, modules, or documentation tiers relevant to the current task	Token waste. Attention dilution. Agent fixates on irrelevant code.
Let agents make architecture decisions	Provide architecture decisions as input constraints, not as questions for the agent to answer	Agent picks a reasonable-looking pattern that conflicts with your system's trajectory. Expensive to undo.

Don't	Do	What Goes Wrong
No iteration limits	Set explicit retry bounds (e.g., "attempt implementation max 2 times, then stop and report")	Runaway loops. Agents that "keep trying" consume tokens and produce increasingly divergent output.
One agent does everything	Separate planning, implementation, testing, and review into distinct agents with distinct context	Role confusion. Planning considerations leak into implementation. Test quality drops when the same agent wrote the code.
Skip the planning agent	Always run a planning pass before implementation, even for "simple" tasks	Implementation without a plan is a spec-free zone. The agent guesses scope, structure, and boundaries.

Process anti-patterns

If you're serious about the automation, entirely possible to achieve with modern AI tech, you need to be serious also about the process. Here are the pitfalls I see most often.

Don't	Do	What Goes Wrong
Set up everything at once	Build capability incrementally: start with project context, then docs, then one agent, then gates	Teams overwhelm themselves before seeing any value. Complexity without calibration.
Auto-approve gates	Run gates manually first. Automate only after the team knows what "good" looks like at each stage	"All tests pass" becomes the definition of done. Architecturally wrong code ships because nobody actually looked. Tests may be wrong or insufficient. Subtle bugs slip through.
Let conventions erode silently	Track plan rejection and rework rate. Investigate template and convention drift	Specs that worked last month stop working. Same as accepting always-failing automated tests.
Use private setups	Provide AI environments with visibility into tools, MCP servers, and context used. Share all the setups (skills, instructions, practices). That's the "next level" of AI-coding!	Miss the learning opportunities. You'll become the 'AI support guy' who is supposed to fix the issues while others enjoy the ride.
"Vibe code" through features	Pause after each agent session to review coherence, not just correctness. Stay in the loop.	"Building, building, building" without stepping back. Result: duplicate logic, mismatched names, no coherent architecture.

@ The anti-pattern that cost me the most time was one-shotting: letting the agent attempt an entire feature in a single pass. Despite explicit instructions to work task by task, the agent would regularly try to implement everything at once, touching files it shouldn't, creating dependencies between components that should have been independent, and generally making a mess that took longer to untangle than it would have taken to do it properly. The only fix that stuck was limiting scope to a single task per invocation. Not "please do one task at a time" in the instructions. Literally one task, one run.

Security & governance anti-patterns

This category of don'ts is not only about agentic development, but these apply to all usage of generative AI systems. The more you offload to AI without proper oversight, the more risk surface you expose to having the prod connection strings injected into your codebase or various tool memories.

Don't	Do	What Goes Wrong
Paste sensitive data into LLMs	Define clear data handling policies; provide sanitized test data for agent context	Pasting directly or via context sensitive data into LLMs. Credentials, PII, and proprietary code leak outside your security boundary.
Skip security review of AI output	Secure software development was hard before AI and certainly remains so. Have experts early on to review. Integrate automated security scanning. Treat all generated code as untrusted input	The code looks right but isn't. Hard security audit fails and prevents rollout. Sensitive customer or company data is exposed or you get hacked.
Ignore code or tool origins	Review licensing implications. Never allow npm install or similar without checking.	Malicious packages. License violations. Unintended dependencies that become attack vectors.
Let AI mask skill gaps	Require developers to explain generated code before it merges. Invest in training alongside AI adoption	Nobody understands the generated code. When it breaks, nobody can fix it. The team has a dependency, not a capability. Everybody ends up in the dumb zone.
"The AI wrote it" as excuse	Treat AI output exactly like any other output for review and accountability purposes	Accountability evaporates. Quality drops. Defect ownership becomes a blame game.

Don't	Do	What Goes Wrong
"Let an agent replace your cloud engineer"	You still need people who know the technology and its sustainable usage	The IAM/IDP/secret management and other security configurations are not something you can just "ask the agent to do". You need to have the people who understand the security implications and how to properly set up the guardrails. And if you're doing IAC, you'll also risk getting an eyewatering bill from Jeff or Bill next month.

Building your software factory

So how to get started with the factory? The following sections suggest a plan to follow to build it gradually, applying the same incremental approach from [Chapter 8's](#) Adoption Ladder. The point here is not that exactly these seven layers are the gold standard (obviously) but illustrate a logical progression of how I've tried to do this. It's best to build on a strong foundation (solid project context, good documentation structure, task tracking) before jumping to parallel agents runs coding the new major ERP (would be good riddance btw) in days.

Building the Machine

Seven phases of incremental capability — each builds on the previous

FOUNDATION

1

Custom Instructions — Project context that every agent respects
CLAUDE.md or equivalent with conventions, stack, constraints

2

Documentation Hierarchy — Structured context agents can navigate
Guidelines, architecture decisions, patterns, domain context

STRUCTURE

3

Task Tracking & Gating — Design the gate state machine on paper
Task granularity, status transitions, gate criteria

4

First Agents — Start with planning, expand from there
Planning agent first, then testing, review, deployment

CALIBRATION

5

Manual Gating & Review — Calibrate before you automate
Human reviews every gate — learn what "good" looks like

6

Automated Gating — Encode what you learned into guardrails
Static analysis, architecture rules, convention checks

MATURITY

7

Skills & Tooling — Reusable capabilities that "just work"
Scaffolding, transformations, validations — continuously refined

Seven phases of building governed agentic delivery: from project context to reusable tooling.

All this is obviously going to be iterative. The tools evolve, models and people change, your codebase and documentation will evolve, and your understanding of what belongs in the project context and how to specify work will improve. So in parallel with this regular process you have for your factory, you need to have the maintenance process for the factory.

So what kind of things to look out for and learn about? I tried to summarize this in the next image.

Anatomy of the Software Factory

Building blocks and their relationships — aligned with the phased approach

Agent = **Model** + **Specification** + **Tools** + **Context**

📁 **CONTEXT** — what the agent reads

Custom Instructions **P1**

CLAUDE.md, copilot-instructions.md — project-level conventions, stack, constraints

Documents **P2**

Architecture decisions, patterns, conventions, domain context — hierarchical & navigable

Prompts **P3**

Task specifications, user stories, acceptance criteria — the specific work to be done

↓
LOADED INTO
▼

⚙️ **AGENT** — the execution unit

P4

Model

The engine — Opus, Sonnet, GPT, Gemini. Matched to the agent's task type.

Agent Specification

The recipe — role, workflow, entry/exit criteria. An .md file with thin yaml frontmatter.

Tools

← Skills

The hands — file ops, search, build, test, deploy. Extended by reusable Skills.

↓
OUTPUT REVIEWED BY
▼

🛡️ **GATING** — quality control

Manual Review **P5**

Human-in-the-loop at every gate — calibrate before you automate

Automated Gates **P6**

Static analysis, architecture rules, convention checks — encode what you learned

↓
ACCELERATED BY
▼

📦 **Skills & Reusable Tooling** — packaged capabilities that extend agent Tools

P7

The building blocks of a software factory and how they relate.

Let's do a quick recap of the concepts here before going into the phases. The core building blocks of your software factory are:

Element	Purpose	Example
Custom instructions	Project ground truth always in context	CLAUDE.md with conventions, stack, and constraints
Documents	Structured guidelines and patterns	API design guidelines, architecture decision records, coding standards
Prompts	The command you or another agent give to agent to start or guide the work	"Create an implementation plan for the next unimplemented user story from feature 34331"
Model	The LLM that generates the output	GPT-4, Gemini, Sonnet, Opus
Agent specification	The recipe for an agent's behavior, including its role, tasks, and context	A YAML/md file defining a "planning agent" that uses Opus and has access to project documentation to analyze and create an clear, detailed plan for a given feature
Tools	The toolset provided by the IDE/Code platform or offered via MCP	Edit file, run agent, find facts from the code, seek documents from web
Gating	The mechanisms to control and monitor agent actions	Approval workflows, automated checks, and validation steps
Skills and reusable tooling	Another way to expose capabilities to models. Economic option for MCP	Custom scripts, reusable functions, and shared libraries that agents can call to perform common tasks, like agent-browser, read a pdf, or do a semantic search on the codebase.

Phase 1: Custom instructions

Start with the "always true" things about your project. A CLAUDE.md, copilot-instructions.md, or equivalent file that captures the conventions, technology choices, and constraints that every agent interaction should respect. Keep this compact. If it grows beyond what fits comfortably in an agent's context window, it's too large. Compare it to your corporate guidebooks: you might have some work place safety rules, stock market rules, GDPR and whatever the European Union has (AI Act, anyone?), but certainly the coding agent won't need to read all of them.

Begin with: brief project intro, perhaps a short vocabulary, technical things like languages, tech stack, documentation locations and such and something generic about the process and architecture. Hard stop things and must-not-do's like "never add any secrets like API keys, connection strings to code" belong here, too.



The first version will be wrong. That's fine. The value of Phase 1 is learning what belongs in project-level context and what doesn't. You'll rewrite this file three times before it stabilizes.

Phase 2: Hierarchical documentation structure

As the project matures, **the single instruction file isn't enough**. Or it will be so big that it eats most of your context window; remember, that event basic system prompts, agent recipe, tool specifications and others are easily **tens of thousands of tokens** alone.

So, design a documentation hierarchy as suggested in this book: divide guidelines, technology decisions, architectural patterns, domain-specific context, and things like standard UI patterns into individual files. Also describe the process and gating once you're ready with that.

Your document library will grow over time, especially now that you have (over?) eager AI-assistants to add more content, so a good index is essential for your agents to find the right context without reading everything. Structure your library in a way it so agents can be pointed at the relevant subset rather than ingesting the whole tree.

See [Chapter 17](#) for the three-tier documentation model (project-level, task-specific, tracking) and practical techniques for organizing context.

Phase 3: Task tracking and gating design

Introduce structured task tracking that agents can consume and update. I suggest having this in place early on, not because of the 100% complete paper trail, but this is really the key piece to get right in order for anything to work properly.

File-based approaches work well for revision control and collaboration. Key design decisions include what granularity of tasks, what status transitions, what gates between phases you want. Consider what integration with existing tools (Jira, Azure DevOps) is required versus what can live in the repository. Also check what ready-made tools there are: I've mentioned [GSD](#), and also [Beads](#) as candidates for this. Or just let the AI generate a tool for you which matches your exact needs.

Phase 4: First agents for selected phases

Don't try to automate every phase at once.

I'd start with planning instead of the coding agent. I know for sure most people think the coding agent is the most tempting. My point here all along was that you'll need this spec-driven thing even if you don't like it, no matter what.

Use models best suited for each agent type (reasoning models for planning, faster or code-oriented models for implementation, UI-focused models for interface design). I've preferred the ever-enthusiastic Sonnet as the orchestrator, GPT Codex for code generation, Gemini for UI-related work, and Opus for planning.

Next, I'd introduce a decent test planner. Start TDD-driven planning early. As with hand crafted code, it's very hard to get good and representative coverage after the fact, especially without proper functional specifications.

Phase 5: Manual gating and review

What this means is that you are in the loop between the major stages - planning ready, coding ready, testing ready, review ready. Once you learn the correct bite size and your codebase is structured and documented, and you are past the project bootstrapping phase, you can ease this.

This phase is where most learning happens. Teams that skip to automated gates miss the calibration period and end up with gates that check the wrong things.

Phase 6: Automated gating and guardrails

Once you understand the failure modes and potential points where things usually go sideways, you may introduce automated checks. We've done static analysis, architecture rule enforcement, convention validation backed by deterministic tools (like static code analysis) and partially AI capabilities.

Define what manual intervention points are genuinely necessary versus what can be safely automated. I'd keep the requirement to review and approve the plan before implementation.

- ✓ I've used a special orchestrator agent for the "YOLO" or "autopilot" mode to coordinate the agents for 'easier' features. But this is only after a few iterations and the groundwork for the project was properly laid. My workflow begins with issuing a planned task, which is delegated to a planning agent, and taken from there if all checks out, and the manual review may be left optional or cursory.

Phase 7: Skills and tooling

Develop reusable skills and command-line tools for operations that should "always work". These are like basic scaffolding, common transformations, standard validations, compilations, better-than-grep semantic search tools, etc. Chances are you'll find excellent ones online (check for instance skills.sh, awesome skills on GitHub, etc.).

These are important building blocks of the agentic software factory which offload repetitive work from both humans and agents and reduce the surface area where things can go wrong. As a summary, check out the diagram below.

Trust and verification

Governed delivery is not a faith-based initiative. If the process is working, specific numbers improve. If they don't, something is wrong: either with the specs, the gates, the agent configuration, or the team's understanding of the boundary between beef and boilerplate.

Four metrics worth tracking:

Metric	What It Measures	What "Good" Looks Like	What Bad Numbers Tell You
Plan rejection rate	How often specs come back for revision at the review gate	Below 20% after the team has calibrated	Specs are too vague, or the team hasn't internalized what "good enough" means
Defect escapes	Issues found in production that should have been caught by gates	Trending downward; no repeat categories	Gates are checking the wrong things, or acceptance criteria are incomplete
Rework time	Time spent revising agent output before it's acceptable	Decreasing over time as specs improve	Spec precision is too low for the story type, or conventions aren't being followed
Reviewer fatigue	Subjective rating from reviewers (survey or standup check-in)	Stable or improving; reviewers feel the output is getting easier to assess	Specs are degrading, agent output is inconsistent, or review criteria are unclear

Plan rejection rate is your leading indicator. It tells you whether the team is getting better at the beef, the upfront specification work that determines everything downstream. Track it weekly. When it drops, the process is calibrating. When it creeps back up, investigate: new team members? Changed requirements patterns? Specification template drift?

@ I don't have hard measurements, but from the early stages to now the chances of getting most things right on the first pass have at least doubled. The specs got better, the context got tighter, and the agent output got more predictable. That said, most of the improvement comes from learning what to specify, not from the tools getting smarter.

Defect escapes is your lagging indicator. It tells you whether the gates are actually catching what matters. If the same category of defect escapes repeatedly (say, integration issues between services), that's a signal to add a gate check, not to blame the agent.

But according to studies the real performance gains are less than we think they are, right? (See [Chapter 3](#).)

i Don't over-instrument. These four metrics can be tracked with a spreadsheet and a weekly standup question. The goal is signal, not surveillance. If tracking the metrics costs more attention than it saves, simplify.

Summary

When all is set, you'll have covered most of the stuff that is needed to get the most out of your agents and avoid the common pitfalls. It'll look something like this (but not limited to).

From Model to Factory

Start small, add layers as you learn

- **LLM Model** — The engine — pick the right one for the task
Sonnet, Opus, GPT, Gemini

receives

- **Prompt** — Direct the model with what needs to be done
User story, bug report, task specification

enriched by

- **Project Context** — Ground every session in project reality
CLAUDE.md — conventions, stack, constraints, patterns

structured by

- **Agent Recipe** — Formalize the role, workflow, and boundaries
agent.md — purpose, steps, entry/exit criteria, scope

equipped with

- **Tools via MCP** — Give agents the ability to act on the world
File ops, search, build, test, deploy, API calls

packaged as

- **Skills** — Package reusable capabilities for consistency
SKILL.md — scaffolding, validations, transformations

orchestrated into

- **Workflows** — Orchestrate multi-agent delivery end to end
Handoffs, gates, state tracking, multi-agent coordination

↑ Everyone starts here

Build up as you learn ↓

The building blocks of agentic delivery, from a bare model to orchestrated workflows.



- The "beef" is the hard, creative work that only you can do. The "boilerplate" is the repetitive, pattern-based work that agents can handle. Don't delegate the beef; it just moves the problem downstream.
- Spec precision should be calibrated to the risk and complexity of the task. Don't under-specify cross-cutting features or architecture changes, and don't over-specify simple bug fixes.
- Avoid anti-patterns in both specification and agent design. Clear, explicit specs and well-defined agent roles are essential for success.
- Build your software factory incrementally. Start with project context and documentation, then add agents and gates one phase at a time. Don't try to automate everything at once.

17

Context Engineering in Practice

Documentation hierarchy, code navigation, and agent memory

Every token spent reading irrelevant code is a token not spent understanding the relevant code, and increases the odds of wrong choices. Every fact an agent fabricates is a fact it failed to look up. So somebody named this dark art of trying to figure the exact intent and material for LLMs this fancy new name, **Context Engineering**. How I see it is that it's about assembling the right information for agents at the right time. Arguably it is both a good token-saving strategy and effective quality enhancer.

| *Next up, Agent Engineering?*

The framework in earlier chapters established the principles: architecture constrains the solution space, specifications narrow what agents can produce, and tokens have real costs.

This chapter is about **how to set up your project so agents can actually find what they need**.

Ground truth: The documentation hierarchy

As we've learned before, software engineering is much more than code. Consider the Inner Circle and Outer Circle metaphor in [Chapter 6](#). Different agents need different kinds of truth at different times, and confusing these leads to either information overload (agent receives everything, attends to nothing) or information starvation (agent guesses because it cannot find the answer).

Three tiers

One way to organize the information relevant to development is to think of it on three tiers:

Tier	Contains	Where It Lives	When Loaded	Update Frequency
Project-level	Conventions, stack, architecture	CLAUDE.md, ADRs, convention files	Session start	Rarely (per decision)

Tier	Contains	Where It Lives	When Loaded	Update Frequency
Task-specific	Stories, criteria, scope, patterns	plan.json, tasks.json, specs	Task start	Per story
Tracking	Status, gates, history	Task tracker, status files	On demand	Continuously

Project-level truths

Here, facts are always true regardless of the current task. These include general architecture, technology stack, coding conventions, naming rules and error handling patterns. They live in files like CLAUDE.md, copilot-instructions.md, conventions documents, and architecture decision records. They rarely change per feature. Ideally they should be loaded at session start and referenced throughout the cycle. But perhaps not: the black-box testing agent might not need to know that much on how the thing works under the hood, but what it is supposed to do, right? It might benefit from detailed testing practices, such as finding or curating proper test data or mocking.

This level of information should be curated first when you start. In small and simple projects, a single document can actually be enough. Just keep it up to date.

Once the thing gets more complex, you cannot and should not put all the things into a single file. Roughly speaking, the further down the standard project context file (such as CLAUDE.md) your principle, rule, or practice is, the less likely it is to be actually followed.

The key is that project-level documents **remain compact and current**. Stale documentation is worse than none at all, because agents trust what they read. It also leads to token waste, which quickly turn into hallucinations. This is information always appended regardless of what you do. The model will treat it as ground truth and ignore what you say in conversation. Keep your AGENTS.md, CLAUDE.md etc short, up-to-date and **relevant for all the workers in your factory**. Do you think your frontend testing agent cares about what kind of database you use? It doesn't and it shouldn't. So don't mention it.

I oncspent a lot of time figuring out why my agent's response language was sometimes Finnish. It turned out the project instructions stated "The UI is in Finnish." No amount of clarification made the effect go away. Moving that to ui.md solved the problem. (Codex 5.2 was the dumbest of crowd)

Task-specific truths and style guides

These are facts that describe the current unit of work i.e. task.

I learned that a good setup for this tier is a structured specification for each story or task, including the functional requirements, architectural constraints, and coding practices relevant to that specific piece of work. For example, a story specification might include: Most of it could be just a **recurring checklist** and a **index of references**, like a list of designs, examples, patterns. Not every things needs to be on pseudocode level.

Category	Examples	Role
Functionality	Stories, acceptance criteria, test cases, business rules	What to build
Architecture	Stack specs, module boundaries, dependencies, API contracts	How to build it
Advisory	Style guides, error handling patterns, reference implementations	The style to build it in

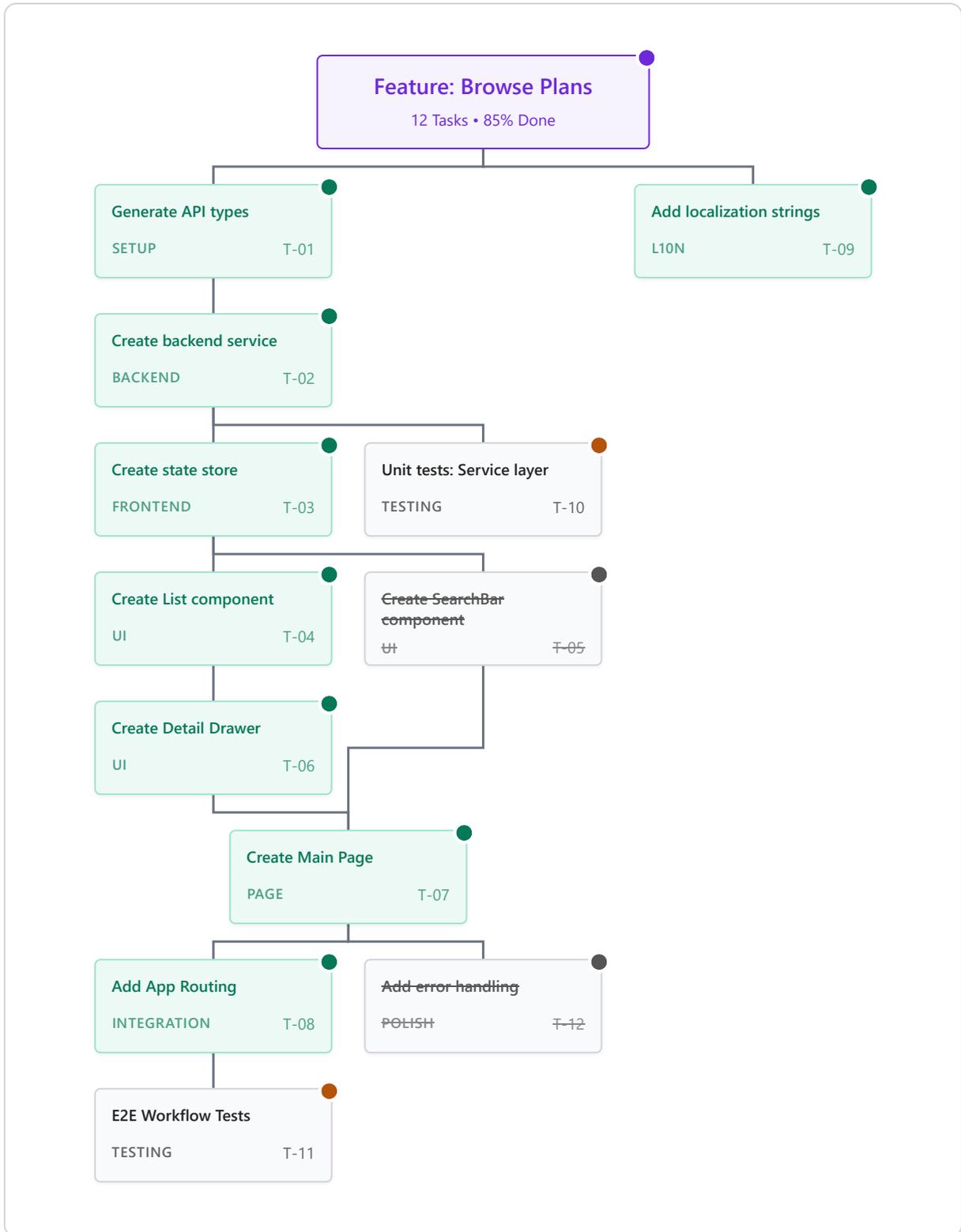
This kind of information is stored in for example `plan.json`, `tasks.json`, story specifications, and linked design documents. Agents load them at task start as instructed, like 'Plan user story 25422'. They form the primary execution context: the instructions the agent actually follows. Or should.

Confluence is a place where information goes to die

Tracking and workflow truths

Short-lived status information, which is not part of the delivery but a development-time concern, should be stored separately from the 'truths'.

This includes metadata about progress and state, how the bigger things have been broken down into digestible chunks, and what has been done to them. Think task status, gate positions, change history, readiness states, review notes and whatnot.



A task breakdown example showing dependencies and state

You may have some of this information linked to your corporate backlog/ticketing system. gets stuck.

I'd steer clear of syncing tickets with Jira and the like during implementation. It's usually enough when something is ready for UAT or gets stuck, not how it's proceeding moment to moment.

Instead, treat tickets as a **communication channel** for PM's and PO's (do we still have those? or Scrum masters?) and the structured planning and tracking files as the execution context for agents.

Agents query them on demand to make workflow decisions, preferably via a good tool and not directly by altering files. Imagine needs such as 'What's left to do', 'What are the active tasks', 'Let me see the plan for this task again'. Needless to say, only the relevant parts of this need to be loaded in the context when executing something.

Indexing, selecting, finding

For token-effective context management, as I suggested by the tiered approach in the previous Sections, the first rule of thumb is to not dump everything into a single context file.

- ✓ The difference between a 200k-token context that is 30% relevant and a 50k-token context that is 90% relevant is not just cost. Irrelevant context not only wastes tokens but actively competes for the model's attention.

So one method I've found pretty good is to organize your documentation into thematic chunks and access them from a compressed index (see [Compressed Index pattern in Chapter 10](#)). It means configuring your agents to seek what they need on-demand rather than pre-loading everything up front. A good index gives pointers to question "where should I look?" without containing all the answers itself.

► Example: Build and test a documentation index

Use AI to create the good hierarchy for you. Use it to detect duplicates, contradictions and gaps.

Create a compressed index. Link it to project instructions or agent specification as needed. Again, let the AI do the heavy lifting.

Review the results.

For example, consider an index that looks like this:

```
`INDEX.md
- API documentation, practices and conventions
  - api.md (contract, examples, patterns, authentication, error
    handling, ...)
- Testing practices
  - mocking.md
...
```

`Next task we have on the list is to implement a new API endpoint for your payment service. To do it properly, your agent needs to find API practices, relevant code examples, and testing patterns for it from the docs.

Trigger a task that should retrieve information from the index.

› **You:** Implement my next task for the payment service.

Inspect the dialogue and tool calls like file reads, and see if the agent is actually using the index to find the right information.

- ◆ **Agent:** Found the next task 'Implement the API'
- ◆ **Agent:** To implement the API, I need to understand the API contract and examples.
- ◆ **Agent:** ToolCall: Read file 'api.md'

If that does not happen (and you got the GraphQL version instead of the standard RESTful that api.md clearly mandates), debug the index structure and your agent specification until it does.

Rinse and repeat. You won't get it right the first time. Use AI to improve itself!

› **You:** Why did you not use the index to find the API contract? Suggest how to fix it.

- ◆ **Agent:** You're absolutely right! Let me investigate and find a fix for this.
- ◆ **Agent:** ...goes on to figure out the index and suggest enhancements for naming/better indexing and whatnot

Sample index structure

I provided an example from a real, large project where we are using Compressed Indexes below.

The copilot-instructions.md file contains the project-level instructions. Instead of embedding all the details, it references an INDEX.md that maps short keys to specific documentation files.

Each agent is configured to embed only the keys relevant to its task to fetch the detailed context on demand rather than browsing the entire document tree.

► Project index and agent cross-reference

copilot-instructions.md:

Documentation Detailed guides in .docs/ – see .docs/INDEX.md for full structure.

Need | Read

API calls & envelope pattern | .docs/01-patterns/api.md

Zustand store patterns | .docs/01-patterns/state.md

UI styling & theme tokens | .docs/01-patterns/ui.md

New page / feature scaffolding | .docs/02-workflows/scaffolding.md

Testing patterns | .docs/02-workflows/testing.md

SSE notifications | .docs/03-domain/notifications.md

Agent workflow phases | .docs/02-workflows/agent-phases.md

Skills reference | .docs/04-skills/reference.md For the engineering agent, we've created a specialized compressed index in engineering.agent.md:

```
`## Documentation
```

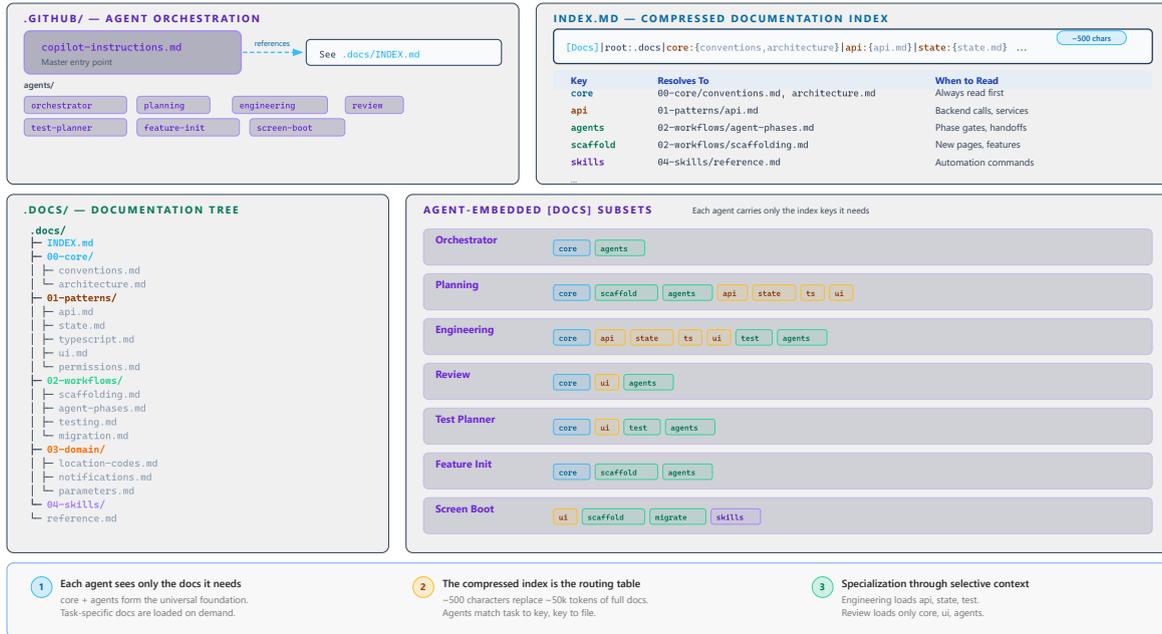
```
[Docs]|core:{00-core/conventions.md,00-core/architecture.md}|api:{01-patterns/api.md}|state:{01-patterns/state.md}|ts:{01-patterns/typescript.md}|ui:{01-patterns/ui.md}|test:{02-workflows/testing.md}|agents:{02-workflows/agent-phases.md}
```

```
`
```

To illustrate the examples and ideas above, check the image below to find mappings between agents and indexing. The project instructions (the 'always true' ground truths) reference the INDEX.md, which maps to specific documentation files. Each agent embeds only the keys it needs to fetch detailed context on demand.

Documentation Index: How Agents Find What They Need

copilot-instructions.md references INDEX.md, which agents use to navigate .docs/



A real-world documentation index: copilot-instructions.md references INDEX.md, which maps short keys to documentation files. Each agent embeds only the keys it needs.

i How these indexes work in practice is crucial for understanding the efficiency and accuracy of the agents. For instance, Claude Code does not like the format and prefers a different structure.

The "Lost in the middle" problem -- do models actually work like us?

Let's imagine you are renovating your bathroom.

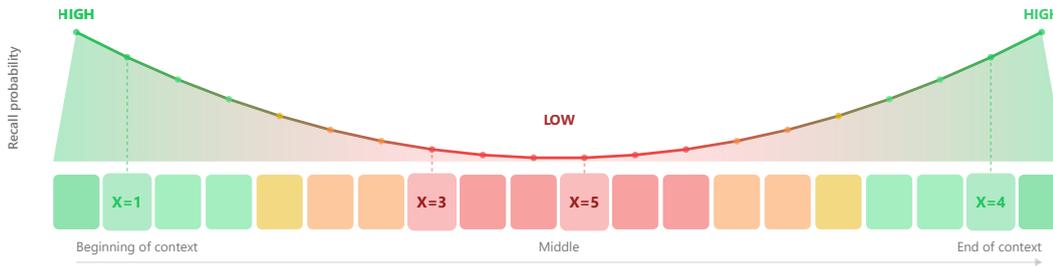
Before getting it done, you've had a long, winding, back-and-forth discussion with a renovator about options: tiles, colors, cabinets, and whatnot. As you guessed, nothing is properly on paper, at least not up to date.

Then you make a verbal contract with him to renovate your bathroom. What are the odds he'll do what you wanted unless you keep very close tabs on what he's up to?

Let's consider this in the context of AI agents and getting a simple thing right, like what is the value of X.

Lost in the Middle

How position in context affects which instructions the model actually follows



What the model is likely to follow:

- X = 4** 38% recency + high attention
- X = 1** 35% primacy + high attention
- X = 3** 15% mid-context, low attention
- X = 5** 12% dead center, lowest attention

Instructions at the start and end of context are followed most reliably.

Information placed in the middle competes for attention and is often overridden by what came first or last.

In the example above, for the model to reliably always say that $X=4$, it would need to work strictly start-to-end on the context and correct itself. That would mean everything later automatically overrules what was said earlier. This is not feasible for several reasons: the high priority for things early in the context is there by design. Otherwise, project ground truths (and the hidden system prompts) wouldn't be followed.

Too much detail and conflicting advice turn into bad results. AI model attention mechanisms have been tuned to prioritize the beginning and end of the context. In a way, it works like humans: you might remember how it all started and ended, but not so much about what happened in between.

So, coming back to your bathroom example, the renovator might remember the initial discussion about changing the tiles to red, but later you said green. He might very well be installing the red ones as we speak after juggling the options and the hardware store and deciding not to call you to clarify.

This is about probabilities, not black and white. You might get lucky that whatever greedy or probabilistic mechanism picks the next token gives you the last value of X , but you cannot depend on that. The model picks one of the values in the $X=?$ set, and it's never 100% certain which. There's deliberate randomness in this.

And don't get me started about the effects of "compacting" indexes.

Finally a couple tips:



1. Condense long sessions to a new 'prompt' by summarizing the previous discussion. Save that to a file, read it, and attach that to a new session. See also the "Dumb zone" in [Chapter 2](#).
2. Try to make your agent to loop over a task exactly 100 times. Most give up, no matter how you formulate the prompt around 10 attempts. I'd be delighted if somebody figured out how to make that work reliably.

Beyond documents: Navigating the codebase

How an agent searches code (or documents) determines how many tokens it spends and how grounded its output is.

In the early days of AI-assisted coding, you'd pick the files you wanted to edit or use first, then fire your prompt. Essentially you were the *context engineer*. Now that we're using agents, that's neither viable nor necessary, unless you decide to split tasks at code-file-level increments and micro-manage your agent tasks.

For all programmers it is pretty obvious that simple text searches do not capture the relationships between your code modules (files) very well. Think programming concepts like inheritance, macros, if-then-else constructs, and whatnot. It gets subtle; imagine you detect a relationship between two modules that one invokes a method on another. While that is a good observation, it might not be relevant to the case in hand at all.

What you need is something called **semantic search**. It is a more intelligent way to find the relevant information, and it can be implemented in different levels of sophistication. In a way the difference between text and semantic searches can be likened to reading every page of a book instead of using the index, cross-references and footnotes.

- i** Semantic search understands meaning and relationships in code, not just text patterns. It's the difference between finding every mention of "PaymentService" and finding the definition, its implementations, and what actually calls it.

In the following, I dig deeper into this topic by looking at three levels of code search sophistication, from basic text search to indexed search to LSP-powered search.

Level 1: Text search (grep and find)

This is the naive baseline strategy you'll see if you look at the tool calls your agent makes. It's often just pattern matching on file names and file contents, perhaps with some synonyms thrown in. The larger the codebase gets, the more inefficient this becomes.

Aspect	Details
Strengths	Universal, zero setup, works in any language, fast. Every CLI agent has access to text search out of the box.
Weaknesses	No semantic understanding. It cannot distinguish a function definition from a comment that mentions the function name. It cannot follow type hierarchies, resolve imports, or understand that <code>PaymentService</code> and <code>IPaymentService</code> are related. It returns noise.
When it is enough	Small codebases, unique identifiers, simple lookups where the string is distinctive enough that false positives are rare.
The failure mode	An agent greps for <code>handlePayment</code> , gets 47 matches across tests, mocks, comments, and the one actual implementation. It reads them all — burning tokens — and sometimes picks up patterns from the test mock instead of the real implementation. The agent does not know which match is authoritative.

i Some tools use better indexing under the hood to cache things and build indexes. But it's still not semantic or aware of actual code structure, call paths, and hierarchies, not to mention spatial relationships in UI.

Level 2: Indexed search, the structural middle ground

A step up from text search: a tool that maintains a code graph. It knows where symbols are defined, what imports what, what calls what. Tools like Serena, tree-sitter-based indexers, and symbol databases fall into this category.

Aspect	Details
Strengths	Structural awareness. It knows that <code>class PaymentService</code> is defined at file X line Y. It can answer "what imports this module?" or "what files reference this type?" It returns the definition, not every mention.
Weaknesses	Requires indexing infrastructure. The index may lag behind recent edits. It does not understand runtime behavior — it knows the static structure but not what actually executes.
When it shines	Medium-to-large codebases where text search produces too much noise. The index cuts through the noise by understanding code structure, not just text.

Level 3: LSP, or how your IDE does it

The Language Server Protocol gives agents the same intelligence that powers VS Code, IntelliJ, and other IDEs: go-to-definition, find-all-references, type hierarchy, rename refactoring, signature help.

Aspect	Details
Strengths	Full semantic understanding. It follows types through generics, resolves interface implementations, understands method overloads, and can trace execution paths through the type system.
Weaknesses	Language-specific — you need a server per language. Setup overhead is real. Not always available in CLI agent toolchains, though this is changing.
Current state	IDE-embedded agents (Cursor, Windsurf, GitHub Copilot) get LSP access for free — the IDE already runs the language server. CLI agents (Claude Code, aider, Codex) are catching up. Claude Code's MCP-based LSP integration is one example of bridging this gap.

Summary

The previous can be summarized as in the table below. Perhaps the key lesson here is that the more accurate information you hand out or prepare right off the bat, the faster and more 'intelligently' your AI coding team will perform.

Dimension	Text Search (grep/find)	Indexed (Serena, tree-sitter)	LSP
Setup cost	None	Moderate (indexer config)	High (language server per lang)
Semantic depth	None — literal text	Structural — symbols, imports	Full — types, interfaces, flows
Token efficiency	Low — returns noise	Medium — returns definitions	High — returns what is needed
Language support	Universal	Broad (tree-sitter grammars)	Per-language server
CLI agents	Default everywhere	Emerging	Limited but growing
IDE agents	Available	Often built-in	Built-in

Confused? Don't worry. This is a lot to digest. I've had only limited success (especially with skills) getting the tools used systematically. Sometimes the agent just does whatever the hell it wants, totally against the instructions, but on a new session it acts perfectly.

I'll conclude here with a couple of practical tips to make your agents use the proper tools (like Serena):

1. Have the absolute minimum set of tools available 2)
Use the exact names of tools explicitly

So spelling and capitalization indeed matter with AI? They do. Don't believe 'em.

2. Start with a clean history and a fresh session
3. Constantly debug and review the skill usage in fresh prompts without referring them directly by name. If they don't get picked up, re-visit your tool/skill description yet one more time.

i Most CLI agents today default to text search. This is like navigating a city by reading every street sign instead of using a map. It works for small towns. It fails for metropolises. As codebases grow, the search technique matters more than most teams realize, and the teams that set up indexed or LSP-based search for their agents see immediate improvements in both token cost and output quality.

Context composition: The assembly recipe

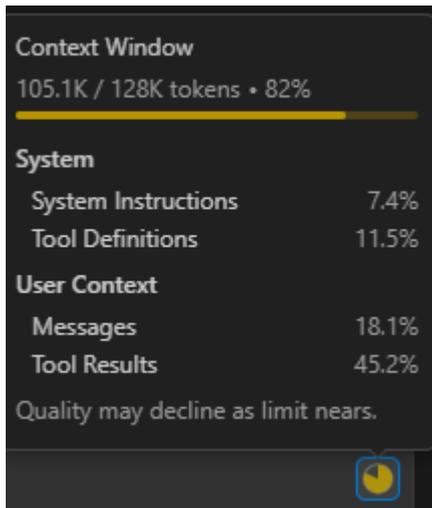
Finding the right information is half the problem. Assembling it into an effective context (in the right proportions, in the right order) is the other half.

The context budget

Treat the context window as a budget to allocate deliberately, not a bucket to fill:

Category	Budget Share	Contains	Risk if Over-Allocated
Project conventions	5-10%	CLAUDE.md, style rules, architecture	Crowds out task-specific detail
Task specification	15-25%	Plan, acceptance criteria, scope	Leaves too little room for code
Relevant code	40-60%	Module code, interfaces, examples	Dominates attention, may include noise
Reference/examples	10-20%	Similar implementations, patterns	Agent may copy rather than adapt
Conversation history	5-15%	Prior turns, corrections, decisions	Accumulates stale context (rot)

The key insight: if code fills 80% of the context, the specification that tells the agent *what to do* with that code gets only 20% of the model's attention. Every piece of irrelevant code you load pushes the instructions further from focus. Budget deliberately.



A real context window at 82% capacity. Tool results and messages dominate; system instructions get less than 8%. This is what 'context pressure' looks like in practice.

Pre-loaded vs. on-demand context

Two strategies for getting information to agents:

Pre-loaded: give the agent everything it might need up front. Simpler to set up, no tooling required. Works for small tasks with predictable context needs. The risk: context bloat. When you pre-load "just in case," you fill the budget with material the agent may never need, crowding out what it does need.

On-demand (tool-based): give the agent tools to fetch context as needed. The agent decides what to look up (using grep, file reads, indexed search, or custom tools). This scales to large codebases where pre-loading is impossible. The risk: the agent may fail to look up what it needs, or waste tokens on dead-end searches before finding the right file.

Custom MCP tools can bridge the gap. Instead of generic "read file" tools, expose project-specific lookups: "get the API contract for service X," "show me the test patterns for this module," "what are the conventions for error handling?" These tools return exactly the right context for a specific question, without the agent needing to navigate the file system.

- ✓ The best approach is usually hybrid: pre-load the always-needed context (project conventions, task specification) and give tools for everything else (code navigation, reference lookup). This keeps the base context lean while ensuring the agent can find what it needs when it needs it.

Limiting visibility: Constraining the search space

The best search result is the one you never had to filter out. Before improving how agents search, reduce what they have to search through.

Monorepos and multi-project codebases

A 500-package monorepo: the agent implementing a payment feature does not need the email templates, the admin dashboard, or the legacy migration scripts. But without scoping, every `grep` command searches all 500 packages, and every "find references" returns matches from code the agent should never have seen.

Scoping mechanisms exist in most agent toolchains: workspace configuration, `.claudeignore` patterns, context boundaries in agent configuration files. The key is using them proactively, not reactively.

This connects directly to [Chapter 11](#)'s module boundaries: good architecture *enables* scoping. If payment and email are cleanly separated packages, you can hand the agent only `packages/payment/` and its shared interfaces. If they are tangled in the same module, scoping is impossible without also untangling the code.

Search scope as a governance lever

The leash concept from [Chapter 10](#) (defining what agents can and cannot do) extends naturally to information access. Scope is a leash on attention, not just on action.

- **Exclusion patterns:** "do not look in legacy/, vendor/, generated/, node_modules/"
- **Inclusion patterns:** "only look in src/features/payment/ and src/shared/types/"
- **Functional scoping:** "you are working on the API layer; the frontend is out of scope for this task"

The need-to-know principle

Start tight, expand on failure. If the agent says "I cannot find X," widen the scope. This is better than starting wide and hoping the agent filters: pre-filtered context is cleaner than post-filtered context, because the agent never sees the noise in the first place.

Scope Level	Example Size	Token Estimate
Full repository	500 packages	~10M tokens (unreadable)
Scoped packages	3 relevant packages	~2M tokens (too large for one context)
Relevant modules	8 modules	~500k tokens (fits with compression)

Scope Level	Example Size	Token Estimate
Specific files	15 files	~50k tokens (effective working context)
Relevant functions	Key interfaces + implementation	~5k tokens (surgical precision)

Each scoping level is roughly an order of magnitude. The difference between a full-repo context and a scoped working context is the difference between a library and a focused briefing.

-  **Scope before you search.** Configure agent visibility BEFORE the task starts, not during. A well-scoped agent finds what it needs in 2-3 lookups. A full-repo agent burns 15 lookups reading irrelevant files before finding the right one, and by then, the context window is polluted with noise from the wrong files.

The case for indexed code navigation

Most agent harnesses still discover code the hard way: grep for a pattern, read the file, grep again, read another file. On a 79K-line TypeScript codebase, finding a single function definition with grep returns 40 matches across 19 files at a cost of roughly 1,500 tokens. A semantic search for the same thing returns 26 snippets at 4,000 tokens. Most of that output is noise.

An indexed code graph changes the economics completely. I ran a side-by-side comparison using a lightweight Rust-based indexer exposed via MCP. It keeps the entire codebase's symbol table, call graph, and dependency relationships in memory, so lookups are structural rather than textual.

Here is what a single indexed lookup actually looks like. The agent needs to find where

`ApiResponseHandler` lives and what it calls:

```

Input: { "name": "ApiResponseHandler", "include_body": false }

Output:
  file: src/utils/apiResponseHandler.ts
  kind: Function
  line: 57
  sig: export function ApiResponseHandler<T>{
  callees: [tryExtractErrorDetails, extractErrorMessage,
           ApiError, isValidEnvelope, isNotFoundError]

```

That is the entire response. 130 tokens. The agent now knows the file, the signature, and every function it calls, without reading a single file. A grep for the same symbol name returns 40 matches across 19 files at 1,500 tokens, and the agent still needs to read the file to get the signature.

Finding all references: 80 tokens instead of 800. Over a typical 15-query exploration session, the indexed approach used roughly 10,000 tokens versus 35,000 for the standard tools. That is a 70% reduction in context consumption for the same information.

Task	Indexed lookup	Standard grep/search	Savings
Symbol definition	~130 tokens	~1,500 tokens	91%
Find all references	~80 tokens	~800 tokens	90%
Full body + dependencies	~1,200 tokens	~3,200 tokens	62%
Complex navigation (chained)	~1,500 tokens	~4,500 tokens	67%
File reading (compressed)	~2,000 tokens	~3,200 tokens	37%

The point here is not that one tool is better than another. It is that the default code navigation approach baked into most agent harnesses is remarkably wasteful. Every unnecessary token spent on finding code is a token not available for understanding and generating code. Tools like Serena (which exposes an indexed code graph via MCP) and CodeGraph (which builds a semantic knowledge graph for symbol lookup, call tracing, and impact analysis) represent a growing category of context augmentation tools that sit between the agent and the codebase, turning brute-force discovery into structured retrieval. [Chapter 18](#) lists more of these in the tool ecosystem overview.

! The default way agents discover code is the most expensive part of most sessions. Indexed, structural code navigation is not a nice-to-have; it is the single highest-leverage optimization for context efficiency.

Agent memory: Persistent context across sessions

The tool builders have heard our cries and moods, and figured it might be better to carry some lessons learned from session to session via so-called memories. Perhaps it is a kind of reinforcement learning, but either way, it can also lead to funny side effects as the memory is just another thing that gets or might get added and be up to date. Let's dig into that a bit more.

How agent memory works

Built-in memory tools. Some agents offer explicit "remember this" capabilities. Claude Code maintains memory files that persist across sessions. Cursor has context notes. These are the simplest form of persistent context: the agent or the user writes something down, and it is available next time.

Custom memory. Project-specific memory files, decision logs, and "lessons learned" that agents read at session start. More structured than built-in tools. These might capture architectural patterns that

worked, common mistakes to avoid, or domain knowledge that is not in the documentation but emerged during development.

Conversation-derived memory. Summaries, key decisions, and corrections extracted from past sessions and stored for future reference. Some tools automate this; others require manual curation.



Memory can rot just like context. A memory file that accumulates stale patterns, superseded decisions, or one-off workarounds becomes a source of hallucination rather than prevention. Memory needs curation: periodic review, pruning of outdated entries, and clear separation between "still true" and "was true once." An unpruned memory file teaches the agent to follow patterns you have long since abandoned.

Practical memory patterns

- **Keep memory files small and thematic.** A `conventions.md`, an `architecture-decisions.md`, a `lessons-learned.md`, not one massive file that mixes everything.
- **Version memory alongside code.** If it matters enough to remember, it matters enough to track in revision control. Memory is part of the project, not the tool.
- **Distinguish project memory from personal memory.** Project memory (shared, in the repo) captures team-wide knowledge. Personal memory (tool-specific, per developer) captures individual preferences and workflow patterns. Conflating them creates confusion.
- **Review memory files during retrospectives.** They are a leading indicator of where agents struggle. If the memory file says "always check for null in the payment service," that is a signal the payment service has a design problem worth fixing.

The compound effect

In this chapter, I've discussed five key concerns for ensuring agents find what matters: documentation hierarchy, search technique, context composition, scope limits, and memory.

Applied carefully, they give your agents a much better fighting chance to one or N-shotting the task at hand without excessive iteration, or starting over.

Good docs with smart search, deliberate composition, tight scope, and curated memory minimize token usage and maximize grounding. Any one of these failing often undermines the others. For example, perfect architecture docs accessed with blind grep over all docs wastes tokens and gives too much context. A great LSP integrated into your flow on an unscoped monorepo still drowns in noise. Finally, clean context with polluted memory will drift toward abandoned patterns and incorrect assumptions.

In a way this is just information management, like an internal search engine. Perhaps this universal project semantic search could be implemented with a good product (and very well may be once things settle).

! Spend time figuring out what kind of project-level truths, feature specifications and targeted instructions you have, and how they should be organized. Then set up your agents to find them on demand rather than pre-loading everything. Use indexing and scoping to limit the search space, and consider memory as a way to carry lessons learned across sessions.

18

The AI Tool Ecosystem

What's available, what it does, and how to choose

Choosing the right tools

💡 If it ain't easy, developers won't use it.' Heard that before? Leaving the obvious 'how come you use vi then' aside, one obstacle to cross on our way to an AI-powered nirvana is to convince the development team to actually use the tools, and in order for that to happen they need to have decent Developer Experience, preferably in their favorite IDE.

For Software Development, the toolset and the primary way to use it has been pretty much stable since the late '90s or so. You have an IDE or an editor system to write, compile and test your code. That's combined with scripts and other tools to, for instance, run, test, or manage the running system.

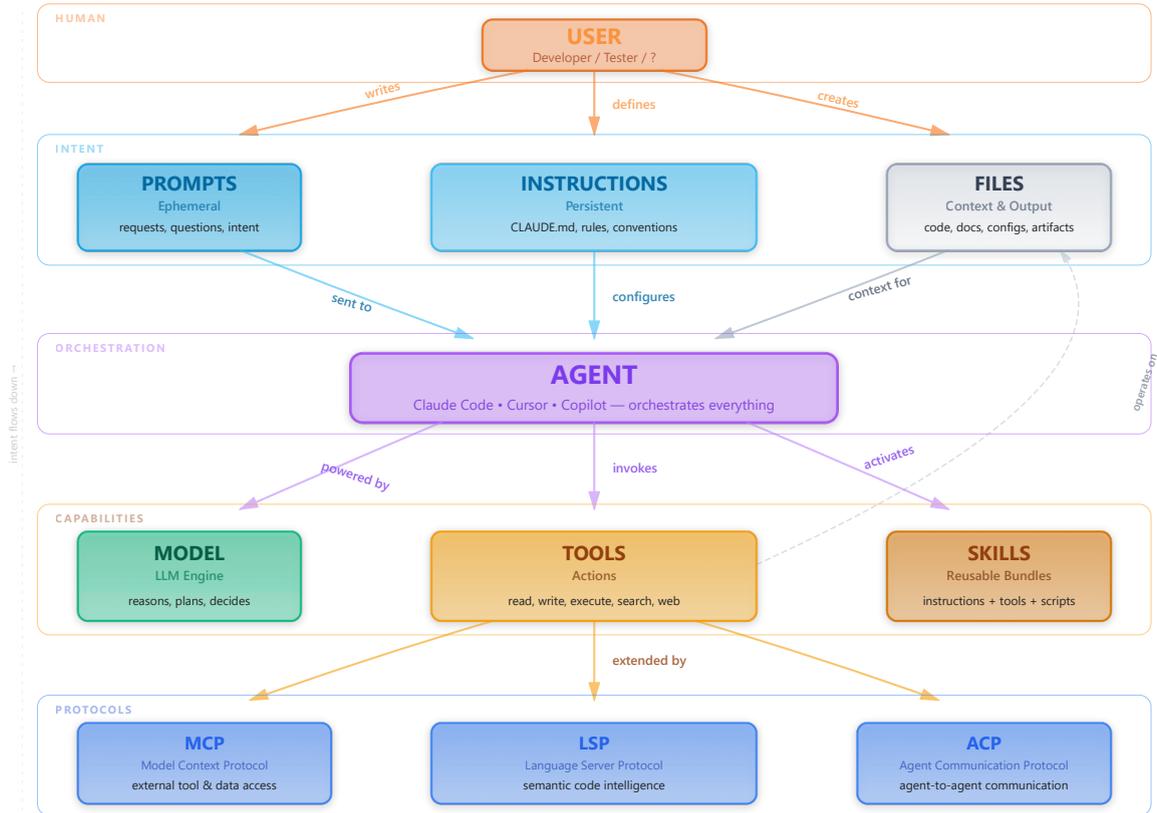
That was the starting point of the AI-powered tools as well: you could (and should) think of them as wrappers between your codebase and the LLMs. A bit like intellisense on steroids running in the familiar IDE of your choice.

I think it is interesting that most of the popular tools are actually Visual Studio Code forks or extensions.

Key concepts related to the current state of affairs are depicted below. Regardless of which tool you use, you prompt, attach context, and some kind of agent processes the request and produces the output.

Basic workflow

Roles of and responsibilities of different actors in an agentic workflow



Agentic harnesses

One way to think about this field is in terms of *agentic harnesses*: the coordination layer that wraps around a language model and determines how it interacts with your codebase, your workflow, and other agents. The harness is what turns a raw LLM into a development tool. Some harnesses are minimal, a prompt and a file reader. Some are opinionated, a full workflow with gates and artifacts. The choice of harness shapes the developer experience more than the choice of underlying model.

Opinionated vs unopinionated tools (generic vs specialized)

Nowadays the AI tools are much more than the generic ChatGPT prompt in the browser we all began with, even though the primary interface is still the prompt.

The primary groups of AI tools in our craft:

- **IDE-integrated assistants** which are embedded directly into your code editor, like GitHub Copilot, Cursor, Windsurf, and Antigravity. They provide in-context code generation, inline chat, and agent mode capabilities.

- **Code- and workflow-focused** tools, often integrated directly into your code hosting platform, like GitHub Copilot.
- **App-centric or user-facing** tools, less focused on the code and infra, like Lovable.
- **CLI tools** which bring us back to the terminal ages, perhaps with somewhat enhanced UX. Claude Code, Opencode, Codex CLI, Amp CLI are examples of this category.

One of the issues with plugging AI capabilities into our craft is actually the unopinionated nature of the tools. They offer you the power of the LLMs, but they don't tell you how to use it. Yes, there are capabilities like tools (retrieval, file access, API calls) and features like agent mode, but they don't tell you when to use them, how to structure your prompts, how to manage the context, how to integrate with your workflow.

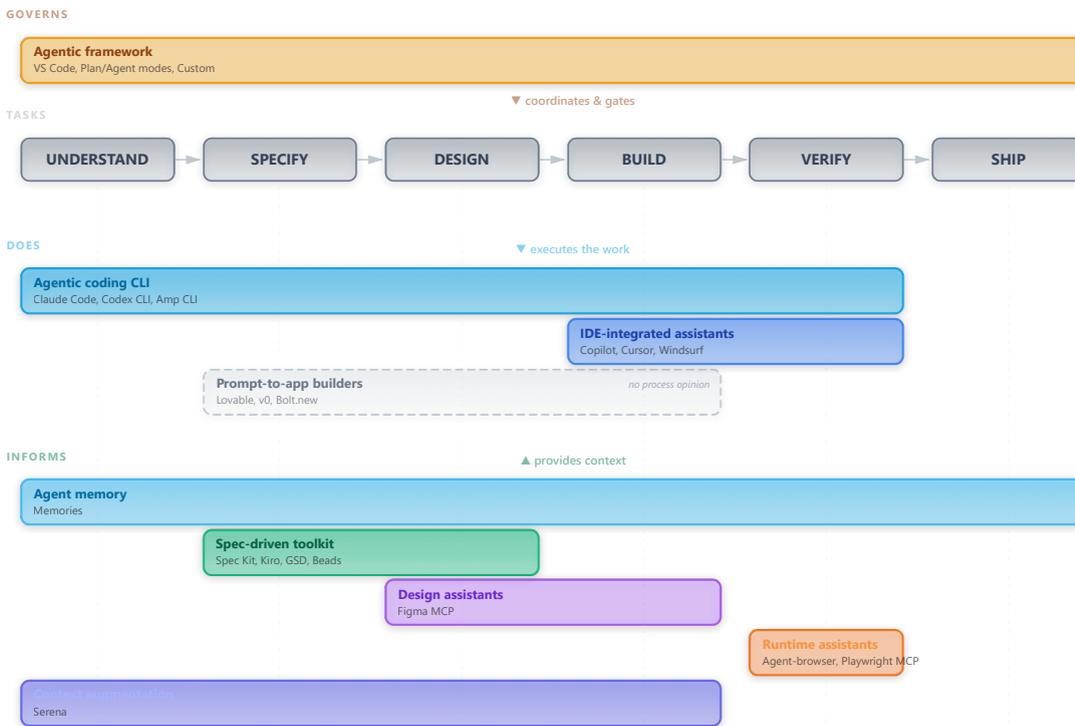
I summarize these distinct tool groups in the table below to compare them with the agentic coordination and context framework proposed in this book.

Category	Tools	Core idea
Agentic framework	VS Code, Plan/Agent modes, Custom	Coordinating layer, parallelism, context management (subagents), handoffs, structured artifacts, human checkpoints
Agentic coding CLI	Claude Code, Codex CLI, Amp CLI	Terminal-resident agent that understands your codebase and executes tasks via natural language
IDE-integrated assistants	GitHub Copilot, Cursor, Windsurf	AI embedded inside your editor — autocomplete, inline chat, agent mode
Prompt-to-app builders	Lovable, v0, Bolt.new	Describe an app in plain English → get a working application
Spec-driven toolkit	GitHub Spec Kit, Kiro (AWS), GSD, Beads	Specification-first workflow: write a spec before code, use it as source of truth
Agent memory system	Memories	Persistent, queryable work state across sessions
Runtime assistants	Agent-browser, Playwright MCP	A runtime view to your application for more dynamic context
Design assistants	Figma MCP	Import and couple your design system with your codebase for better design-to-code handoff and context
Context augmentation tools	Serena	Give access to indexed code graph beyond grepping and reading files

Another way to look at this is in stages: whatever 'agentic framework' you use to coordinate and delegate the work, different kinds of tools and capabilities match certain stages of that flow. This is depicted below.

The Tool Sandwich

How tool categories layer around the development workflow



CLI-based agentic coders

Claude Code, Codex CLI, Opencode, and Copilot CLI are general-purpose LLM frontend engines built on a command line interface. They have artifacts for subagents, different modes (edit/plan), and so forth, and use the same context and configuration management approach as their IDE-integrated counterparts: basic instruction files, skills, prompts, and whatnot.

Calling them CLIs to somebody who thinks that means `ps | awk` rather than a nethack-like window is perhaps a bit misleading. These CLIs are not flow-based (text flowing in your terminal) but use colors and organize the windowing to keep the current screen in focus, with the ability to scroll just parts of the window.

IDE-integrated assistants

Copilot and Cursor fill a similar role: they accelerate coding within a stage. The developer might use Cursor during the Engineering gate for day-to-day work, while the framework handles the broader lifecycle around them. Copilot has evolved from inline autocomplete to full agent mode that can plan multi-step edits and create PRs, but it remains a coding accelerator, not a delivery governance framework.

Prompt-to-app generators

Lovable, v0, and Bolt.new solve a completely different problem for a completely different audience. They produce demos; this framework produces governed deliveries. There is no integration point between them.

These tools are extraordinarily effective at time-to-first-demo, and they are reshaping expectations about how fast software should appear. But every one of them hits a "technical cliff": authentication, security, performance, testing, team collaboration, and long-term maintenance are your problem. The demo is free; production is not.

This matters because these tools set the pace expectations that professional delivery teams are measured against.

I think it's a matter of preference whether you stick with IDE-integrated frontends or terminal tools. In certain respects, the return to 80x25 characters without mouse support and lots of hotkeys feels a bit of a downgrade. If I were to guess, these agent/LLM coding interfaces will remain more as frontends for spawning background processes rather than the primary way to interact with code. What I've seen is that people run these inside IDE windows to see the codebase at the same time.

Spec-driven toolkits

GitHub's Spec Kit shares the conviction that specifications should precede code, but differs in scope and enforcement. Spec Kit plans per-project with advisory checklists; the governed approach plans per-story with enforced approval gates and machine-readable schemas tied to backlog items.

Dimension	Governed approach	Spec Kit
Scope	Per-story, program-level	Per-project, greenfield-focused
Enforcement	Hard gates, state machine	Advisory checklists, AI-interpreted
Human checkpoints	Mandatory approval stops	Optional review suggestions
Backlog integration	First-class (Jira, ADO, GitHub Issues)	Minimal
Agent relationship	Opinionated: role-specific agents	Agent-agnostic, works with any tool
Existing codebase	Core use case (modernization, legacy)	Designed for greenfield projects
Setup effort	Significant: requires custom configuration	Low: works with minimal setup
Best suited for	Multi-team delivery needing traceability	Solo or small-team greenfield projects

Context engineering layers

GSD fights context rot through task-level isolation: each task gets a fresh 200k-token subagent context. The governed approach fights context rot through role-level isolation: each stage is a separate agent invocation with defined inputs. Both work; they optimize for different things.

GSD optimizes for solo throughput with minimal ceremony. Its creator's philosophy: *"I'm not a 50-person software company. I don't want to play enterprise theater."* That resonates for good reason, and my roguelike project (the thread running through this book) was built entirely with GSD. It works, and it works well. Most software is not built by large teams with compliance requirements. For a solo developer or a small team shipping a product, GSD's lightweight phases with optional verification give you structure without the overhead of mandatory gates, formal artifacts, and human checkpoint ceremonies. The ceremony has a real cost: configuration effort, context overhead, and slower iteration cycles. If your project doesn't need traceability or multi-team coordination, that cost buys you nothing.

The governed approach earns its overhead when the work must be auditable, when multiple people need to understand what was built and why, and when the cost of an undetected defect outweighs the cost of a gate. This is not a hierarchy where governed is "better"; it's a spectrum where the right choice depends on your context.

Agent memory systems

Beads provides persistent, queryable work state across sessions, a Git-native issue tracker designed so agents can wake up, ask "what's next?", and resume work. The governed pipeline's task-tracking layer serves a similar function but is inseparable from its gate model: state and workflow are unified in a purpose-built artifact store, versioned in Git alongside the code.

Token economy

The Generative AI models, especially the larger ones, are expensive to run. The hardware that runs them is nothing short of the things referred to as supercomputers just a few years ago. So the slowness you'll see - a simple prompt might take minutes to complete, is not due to business of the service, but the immense calculations required to produce the response tokens.

Comparing token efficiency by approach or tool would be an interesting exercise, but is beyond the scope of this book. Anyway, I argue that keeping tasks small (my key thesis!), having clear separations of concerns and well-defined inputs and outputs, and avoiding unnecessary back-and-forth with the model are the best ways to keep token costs down. In practice this means opening fewer files (like e.g. via Compacted Indexes), having clear task definitions that don't require multiple rounds of clarification, and using the model's output as directly as possible without needing to reprocess it too much are effective techniques not only to save money, but make your agents run smoother and faster.

Choosing what fits

What I've described throughout this book is not a product. It's an agentic SDD workflow, pieced together from available tools: shareable, version-controlled task artifacts; multiple developer intervention points; and automatic quality control with a feedback loop. As of this writing, none of this exists as an off-the-shelf product you can install and run.

So how do you actually choose? Four questions determine your starting point:

1. **How many people touch this codebase?** Solo work needs freedom; shared work needs traceability.
2. **What's the cost of an undetected defect?** A PoC demo crash is a shrug; a production billing error is a lawsuit.
3. **How long will this code live?** A throwaway prototype needs no governance; a system maintained for years needs decisions to be traceable.
4. **Will you be around to maintain it?** If you're building something short-lived, speed over quality is a legitimate trade-off. If you'll be living with this system for years, invest in the structure that makes future-you's life bearable.

Your answers map to the table below:

Your situation	Recommended stack	Why
Solo developer, side project or PoC	IDE assistant (Cursor/Copilot) + good CLAUDE.md. Add GSD or Beads if you want lightweight structure.	Plan and Agent modes get you surprisingly far. Governance overhead buys you nothing here.
Small team (2-5), shipping a product	CLI tool (Claude Code) or IDE assistant + spec-driven toolkit (GSD/Spec Kit) + basic conventions docs	You need shared context and some structure, but mandatory gates and formal artifacts are overkill. Spec-first habits pay off even without enforcement.
Medium team (5-15), professional delivery	CLI/IDE tools + hierarchical docs + task tracking + manual gates + at least planning and review agents	Multiple people touching the same codebase means you need traceability. Start with manual gates and automate as you learn what breaks.
Large team or regulated environment	Full governed approach: role-specific agents, enforced gates, state machine, automated quality checks, audit trail	The cost of an undetected defect or an untraceable decision justifies the overhead. This is where the framework in this book earns its keep.

You'll know it's time to move to the next row when you start seeing things like the same error happening again, an agent keeps spilling output that makes a mess of somebody else's work or

totally disregards your 100kB carefully crafted AGENTS.md, or a reviewer can't tell *why* something was built a certain way (or why it was done in the first place).

I could go on, but these kinds of signals are just **the Universe telling you that you need more structure**.

A few rules of thumb that apply regardless of team size:

1. **Start with context, not agents.** A well-written CLAUDE.md and organized documentation will improve every tool you plug in. Skip this and no amount of agent sophistication will save you.
2. **Add structure when pain appears, not before.** If you're losing track of what was built and why, add task tracking. If agent output keeps missing the mark, add a planning agent and specs. If defects keep escaping, add gates. Don't set up infrastructure for problems you don't have yet.
3. **Match the model to the job.** Reasoning models for planning, fast models for scaffolding and boilerplate, code-oriented models for implementation. Using Opus for everything is like driving a truck to the corner store.
4. **Invest in skills and reusable tooling early.** A good scaffolding skill or a semantic search tool will pay for itself across every project, every team member, and every agent invocation.

This kind of rather strict and multi-stage governance is really only necessary when you have multiple people working on the same codebase, when you need to be able to trace decisions and changes, and when the cost of a defect is high enough to justify the overhead.



Pick the lightest approach that meets your actual needs, and grow it as you go.

19

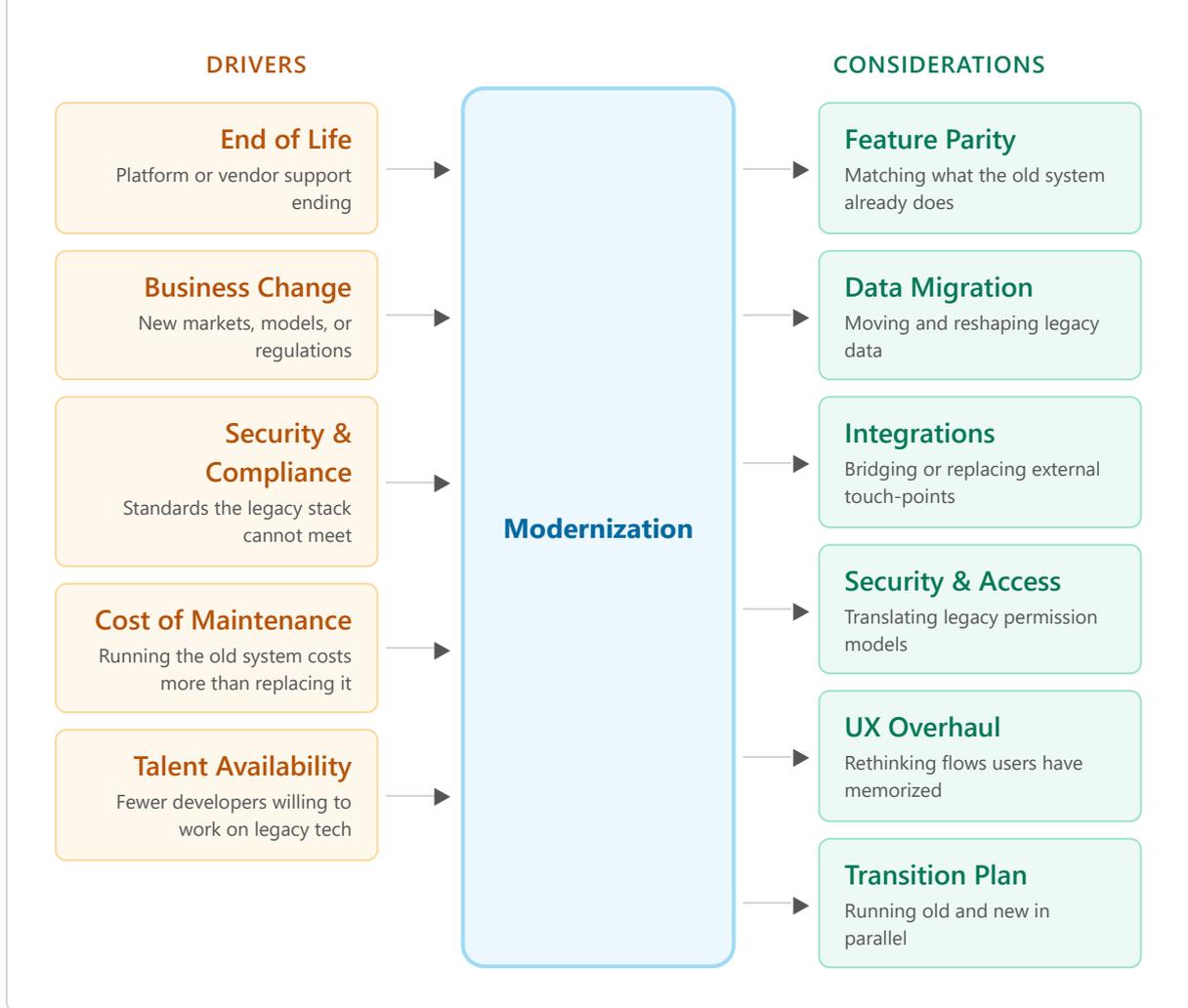
Modernizing Legacy Systems

Turning legacy baggage into a launchpad

More often than not you won't get lucky and be able to start fresh. Greenfield projects are much rarer than brownfield ones, so there's often some baggage to deal with when you start. Perhaps unsurprisingly, there's growing enthusiasm about breathing fresh life into old, possibly outdated systems without the need for a manual, costly rewrite with help of AI. This chapter discusses the common topics relevant to this kind of work and suggests practical strategies for using AI to help you get on board.

Anatomy of a Modernization Programme

What drives the decision — and what you have to get right



Modernization is driven by business and technical pressures, but the real work is in the considerations on the right.

The diagram above depicts some common drivers for modernization and the aspects typically involved. The list of drivers or aspects is by no means exhaustive, and not all of them are relevant in every project.

Next, we'll dig deeper.

Modernization vs greenfield vs brownfield

In the context of this book, I regard *modernization* as a full or partial rewrite of an existing system using modern technologies, patterns, and UX.

What actually gets modernized depends on the case. It could be a partial rewrite of some features, replacing a layer (just the UI, integrations, or some third-party technology), or a complete rewrite. The transition strategy varies just as much.

@ Speaking of modernization — how come I ended up writing a game that looks like it's from the dawn of computing? First of all, it runs on a *browser*, and secondly, many tout the CLI versions of IDEs as 'improvements'.

The goals might be anything from solving technical end-of-life (risk management), security concerns, new business needs, or changes in ownership or hosting. In many respects, modernization is a special case of a greenfield project, but with restrictions and constraints that come from the legacy system.

What a *greenfield* project is should be clear, but for completeness: it's something where you can start fresh without much regard for what came before, if anything.

Many younger devs don't realize that most of your career will be brownfield work. Like a car, it's new for a very short period of time.

A totally new service, system, or app that doesn't replace anything. A *brownfield* is a regular software project you join after it's been running for a while.

Here are my highlights on how a modernization program typically differs from greenfield and brownfield projects across some key dimensions.

Aspect	vs. Greenfield	vs. Brownfield
Starting point	No old code or documentation to worry about	Established patterns and codebase to learn
Documentation and knowledge	Requirements defined upfront ↕ Modernization reverse-engineers specs from code and tribal knowledge	Teams may have incomplete docs but know the system ↕ Modernization often gets little to no documentation
Technical debt and scope	No debt, scope bounded by choice ↕ Modernization is often motivated by the debt itself, and scope is dictated by the legacy system	Debt accumulates gradually, scope is incremental ↕ Modernization must understand the full system even when only rewriting parts

Aspect	vs. Greenfield	vs. Brownfield
Feature parity and UX	Choose which features to build and design UX from scratch ↕ Modernization must replicate existing features and decide what legacy UX to keep, phase out, or overhaul	Evolves features and UX incrementally ↕ Modernization may require a complete rethink while preserving user expectations
Data and integrations	Design schema from scratch, choose integrations freely ↕ Modernization inherits legacy schemas, data migration, and integration points that are difficult to change	Works with existing schema and integrations ↕ Modernization must untangle, migrate, and bridge all of these while maintaining integrity
Rollout and onboarding	Deploy in any order, typical onboarding ↕ Modernization requires careful sequencing plus learning both old and new systems simultaneously	Deploy incrementally, learn one system ↕ Modernization may need parallel deployment and dual-system knowledge

Of course the table oversimplifies a complex topic, but it's worthwhile to understand the starting position and why we need to account for things that aren't relevant in other project types. Now that we have a foundation, let's look at how to get started.

Getting started with modernization

Starting points, motivations, scope and scale of the efforts vary greatly, so it's very hard to give you a 'patent recipe' to tackle this. But here's how I'd go about planning and preparing for a modernization effort. I suggest the following workflow:

Understand the drivers, goals, and phasing

This might seem like a no-brainer, but any vagueness about *why* we're doing this will lead to problems later. What is the primary driver and desired end state? Also, what is the migration strategy and general time plan? It might be far more complicated than it looks at first glance.

Discuss the scope, limitations and general requirements for the modernized solution

Regardless of how you facilitate the discussion, it's important to get a clear picture of what the modernized solution should look like and what the constraints and limitations are. This is the

foundation for all the work that follows, so it's worth investing time to get it right. I've divided this work into the following categories.

Modernization Discovery Areas

Before you write a single line of new code, get clarity on each of these

 **Drivers & Goals**

- › Why are we doing this?
- › What is the desired end state?
- › Migration strategy & phasing
- › Timeline and milestones

 **Scope**

- › Full rewrite or partial?
- › Which layers are affected?
- › Non-negotiables vs. nice-to-haves
- › Architectural constraints

 **Complexity & Features**

- › Feature inventory & user flows
- › Scale: screens, actions, rules
- › Complex workflows & forms
- › Feature parity requirements

 **Documentation**

- › What docs exist today?
- › How current and accurate?
- › Tribal knowledge holders
- › API specs, data models, wikis

 **Code & Technology**

- › Full code access?
- › Can you run & test it?
- › Test coverage & quality
- › Target tech stack & hosting

 **Data**

- › Preserve or start fresh?
- › Schema access & samples
- › Migration scope & quality
- › Data flows & transformations

 **Integrations**

- › How many & how complex?
- › Traffic volume & patterns
- › Specs available? (OpenAPI, WSDL)
- › Replace, bridge, or keep?

 **Co-operation**

- › Stakeholder & user involvement
- › Who drives UX decisions?
- › In-house vs. outsourced?
- › Knowledge transfer plan

A summary of the key areas to investigate before starting modernization work.

In detail, you should be asking the following questions.

Complexity and features:

- Can you produce a list of the features, user flows, integrations, and data flows?
- Can you assess the scale of the system: how many screens, how many actions on each, complex forms or workflows, complicated business rules?

Documentation and knowledge:

- What documentation exists for the current system? User manuals, technical documentation, API docs, data models, WIKIs, etc.
- How up-to-date and accurate is the documentation?
- Are there knowledgeable individuals who can provide insights into the system?

Architectural constraints:

- ? What are the technical requirements the modernized system must meet?
- ? Is there a preference for certain technologies, hosting platforms, IDP, and CI/CD pipelines?

Code, technology and hosting:

- Do you get full access to the code, and to somebody who knows it?
- Is it possible to run the code and test it?
- Can we access a test system to explore and test?
- Does the code have tests? Are they up-to-date and do they cover the critical paths?

Data:

- Are we supposed to preserve the existing data and data flows, or is it possible to start with a clean slate?
- Can we see samples of the data and the schema(s) of the databases?
- What kind of data needs to be migrated?
- Is the data up to date and good quality, or is the migration selective?

Integrations:

- Are there integrations that need to be preserved? How many, and how busy and complex are they?

- Is there an up-to-date specification of the protocols and data formats (OpenAPI, WSDL, etc.)?

Co-operation and involvement:

- Involve the users and stakeholders in this discussion. What are the non-negotiables vs. what can be improved or phased out?
- What is the end-user positioning: are they driving the general requirements, e.g. regarding UX? Will they be engaged in the process?
- Is the customer maintaining the system themselves or is it outsourced? If outsourced, how is the relationship and knowledge transfer going to work?

Collect all materials and interview users, developers and maintainers

Once you've done the initial verification and have a good understanding of the drivers, goals, and scope, it's time to start collecting all the materials and information you can get your hands on. Insist on seeing the old system in action. It's worth a thousand PowerPoint slides. Meet the end users, developers, and maintainers to get their perspective.

Truth is in the code. Always. According to a colleague, "if you have sources you also have full documentation" (he is not the most ardent spec-writer).

If the old code contains a lot of rules and logic, get access to test systems to see the real starting point. There are usually surprises: a "simple" integration protocol turns out to be far more complicated than it looked, a plain VB form is actually a sophisticated calculation engine full of customer business rules, and so on.



You might think that **any documentation is better than no documentation**. It's not. I've been handed documents on continuous paper printed on a matrix printer as the "source of truth" for a legacy system. Judging by the looks, it was probably printed in the early 90s. The year was 2001. Make sure you assess how up to date all material you receive actually is.

Due diligence: assess the state of the old system

Now that you've gathered and assessed everything you could get, it's time for an honest assessment. Depending on the stated constraints, you have choices to make.

The first and obvious one, regardless of your belief in the superpowers of AI, is the GO/NO-GO decision.

If the entire thing is technically well past its prime, you weren't allowed to see any code, data, or the live system, the original developers retired twenty years ago, and there's no knowledge transfer whatsoever, you need to weigh the risk. At a minimum, have contractual guardrails for the surprises that will come after the project kicks off.

Many customers have the healthy tendency to ask for estimates: how much will this cost? For us as suppliers, this is a hard question. Unless you're very familiar with the old system and its technology, think twice before offering anything more than non-binding order-of-magnitude estimates.

Perhaps that's an extreme case, but not unheard of. A healthy assumption, especially with anything written after the Agile Manifesto, is that there's no documentation whatsoever. That's been the case nine times out of ten in my career.

Certain areas require more due diligence than others. If I had to nominate the most important things to look out for, they'd be:

- *Integrations* are often the most expensive and difficult things to migrate. Not only because they're usually the most complex and least documented, but they also involve several parties and systems.
- *Big bang vs. incremental modernization*: be careful with both. You need either a very simple case or very good knowledge of what both strategies would *really* mean.
- *Unused features*: chances are many parts of the old system aren't used at all or are single-user maintenance tools. IT might be unaware of that, so you need a proper walkthrough of everything to verify. Otherwise you'll waste time and effort on something not even needed.

At the end of due diligence, you've made a commitment and decision, and perhaps have already developed an idea of how the programme should commence. Let's look at that next.

Develop a solid, actionable plan

Once you've passed the GO gate for your program, you need a solid plan. I'm not going to turn this into a project management book, but I'll share some key things you'd want in your plan when modernizing an existing system.

- Reserve enough time for careful inventory, research, and requirements analysis. It will pay off. No amount of GPU cycles thrown at reverse-engineering the old codebase will replace this effort.
- Craft a careful strategy for what you expect to get out of the inevitable AI-powered static code analysis. A feature list? Something else? You'll get line counts, function call paths, perhaps some synthesis of them, but that's not going to be the specification of the new system. It'll just reflect the old.
- Legacy code (especially VB forms and the like) might, however, give reasonable entry points, especially if you're just modernizing the UI layer with more or less the same actions and data.

I've had some success with this kind of straightforward conversion, which does not mean 1:1 module or file mapping but taking the old code as a specification source.

The following diagram illustrates this synthesis model. Information from the old display or page you're modernizing and its code-behind are provided as *supplementary context*, useful for understanding what exists, but not the blueprint. The new UX design, architecture decisions, and user input on the right are the actual specification. The agent reads both sides but builds to the right.

Design-Driven Modernization

SUPPLEMENTARY — LEGACY ARTIFACTS



Screenshot

Visual reference of current UI behavior and layout



Old Code

Forms, code-behind, business rules — read, not replicated



Documentation

User manuals, glossaries, domain terminology and workflows

PRIMARY — DESIGN INPUTS



New UX Design

Modern interaction patterns, user journeys, wireframes



Architecture Design

Target stack, API contracts, component structure



User Input

Real needs, pain points, workflows that matter today

Agent-Assisted Synthesis, Planning, Implementation and Testing

◆ Old code as *context* + ◆ New design as *specification*

Synthesis

Understand old, design new

Planning

Specs, tasks, architecture

Implementation

Build to new design

Testing

Verify against requirements

The agent reads legacy code to understand **what exists**, but builds to the **new design** — not a 1:1 translation.

Modern UI

Built from the new design, informed by the old — not a port of legacy semantics

Old code informs, new design decides. The legacy artifacts are context, not specification.

With this approach, I've already transferred some very complicated existing forms to a modern web-based UI. Think late-90s technology with large code-behinds, dozens of tables, filters, popups, parameter screens, and API calls wired together through complicated data structures. The idea was to extract the main data entities and actions from the old code, plan a component structure interactively using our design guidelines and a reference layout, and then forward that plan to the

development and testing agents. What would have been at least a week-long effort not long ago took just hours. All this required careful planning, analysis, and tuning of the agents to understand the task correctly. And yes, a few trials and errors.

- i** Beware of the legacy mirroring risk. Unless something very serious is wrong with the old system (like malfunctioning mainframe hardware), it's rarely worth just converting old code to another language or stack. If you ask an agent to summarize the old code, it'll do it. It won't magically produce a clean specification for a modern system.

Don't just copy the old system

As I've already suggested above, no IT system gets better just by rewriting it in some other language. It will be just as horrible with slightly better UX.

Software engineering, patterns, and architecture have evolved for a reason, and chances are your users' requirements aren't the same as they were 20 years ago. Sure, there have been sidesteps, even very bad ones (hello, CORBA), but things are just *better* nowadays.

So how do you ensure that your modernization effort is actually a modernization, and not just a rewrite of the old system? If there's a single point to take away, it's this:

- ✓** Don't take anything old as a blueprint for the new. Things have changed, and usually for the better.

I sincerely believe that modernization of legacy systems, when orchestrated carefully, can be done at a fraction of the cost it would have taken just a couple of years ago, provided you do it with care and due consideration. AI assistance can make a real difference for organizations struggling with legacy systems. Even building your own solution vs. buying a product or SaaS is a viable strategy again.

- !** Legacy code is context, not specification. Understand what the old system does, then design the new one from modern principles — not from the old architecture.

20

Closing the Feedback Loop

Thinking about the agentic world, there are several different feedback loops. Some of them are familiar and certainly not anything specific to AI-assisted development.

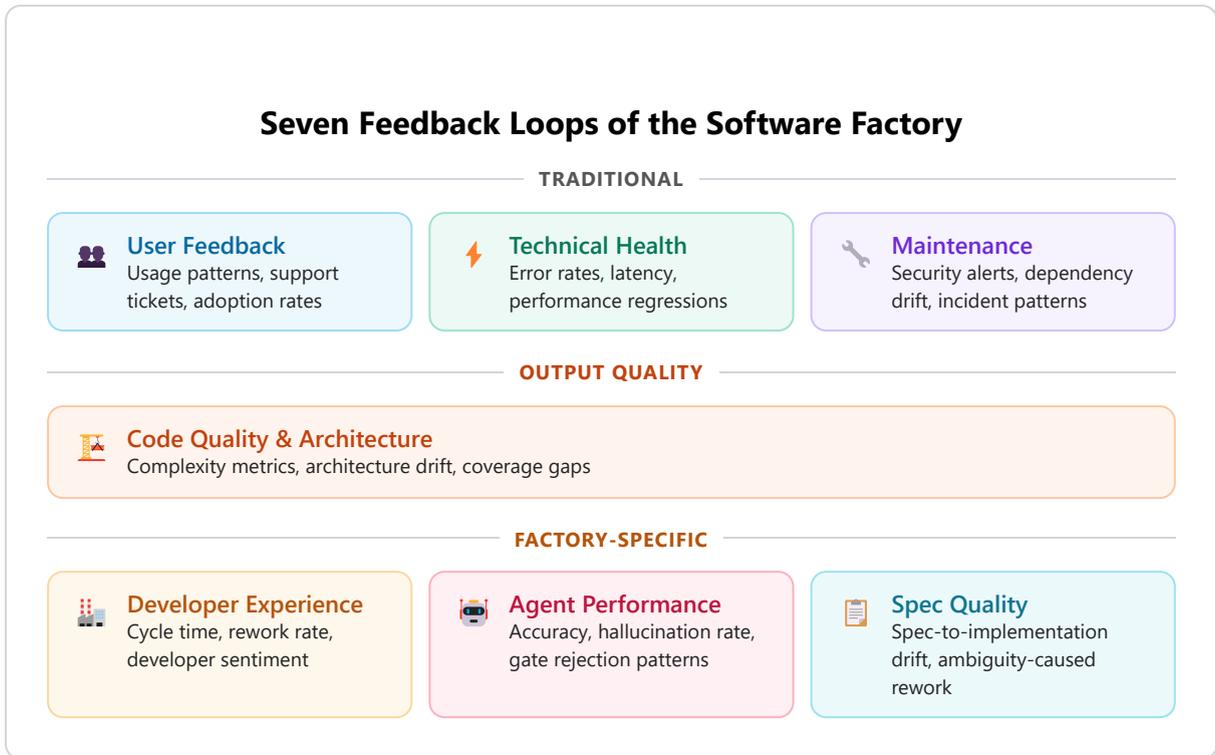
I listed some of the signals that we can collect to improve our process. I'll add a few below, and will discuss some ideas for incorporating them into your factory's tuning process.

- How did the software we developed actually do? How many tickets, what kind of usage patterns there are?
- How did the software perform in production? Error rates, performance regressions, resource usage?
- How do we keep the software up to date, secure and manage it? How do we monitor the software and make sure it continues to meet the needs of users?
- Does the code meet our quality standards and architectural intent? Are we accumulating tech debt faster than before?
- Does the factory work well for the teams and developers? Is there so much to fix afterwards that it becomes more work than was saved?
- How do we monitor our agents in the 'factory' and continuously improve their performance and accuracy?
- How well did our specs capture the intent and guide the development? How do we improve them based on what we learn?

The questions above, and more, are grouped below into two separate boxes:

1. What were the feedback loops that we had before, like user feedback, technical characteristics, and maintenance issues. These still exist, and no doubt we could build automation to improve our next delivery accordingly. Think like DevOps, and perhaps the thing that should've been part of the Agile way (constant improvement).

2. The new one is the 'factory' feedback loop, where the sender or receiver is not a person anymore, but your agent. What often goes wrong? Are people struggling to get the most out of the machinery we have with constant breakdown and bad output? Are we feeding the factory with correct instructions (specs), and are we able to find the perfect level and content for them?



Let's discuss these loops in detail.

User feedback loop

There's a lot we should learn and can learn from the users of our product. Support tickets, on-call feedback, and user sentiment are all important signals that can help us understand how our software is performing in the real world and what we can do to improve it. Some of it is simple fixes: imagine posting an incident to support, and discovering it's been automatically fixed in minutes instead of the usual days or weeks? Or asking for a small feature or change of UX to help you with your job.

The current way to handle issues is typically something like this:

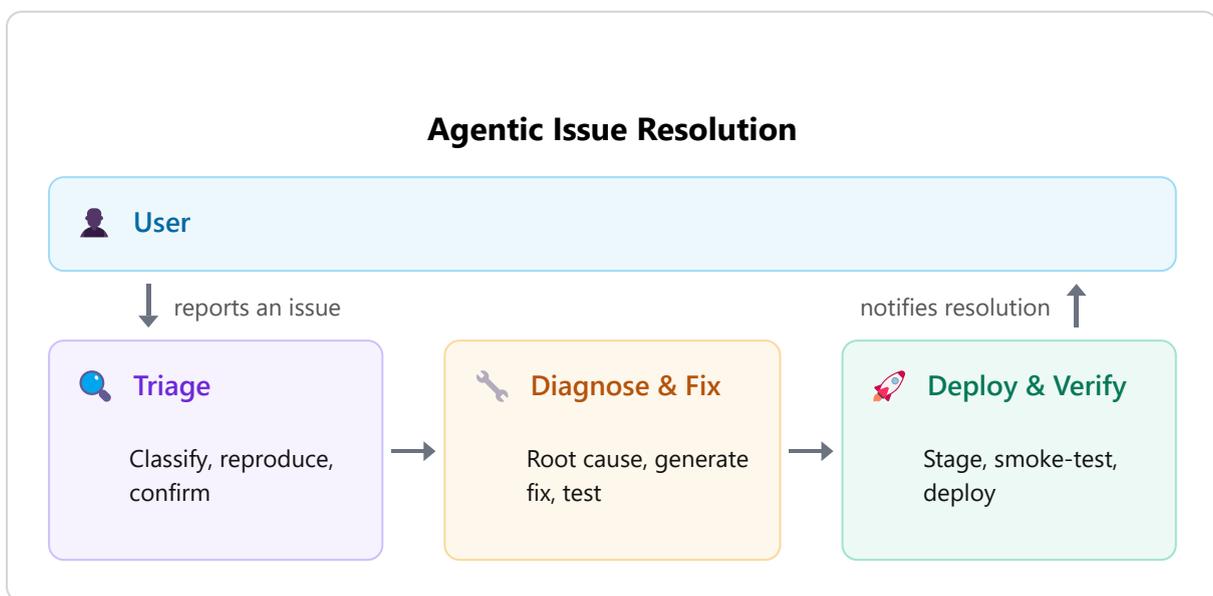
1. User reports an issue through support channels (email, chat, ticketing system).
2. Somebody in the ITILized process triages the issue, assigns it to the right team, and adds it to the backlog.
3. The team prioritizes the issue based on severity, impact, and other factors, and schedules it for a future sprint or release.

4. Somebody does a root cause analysis, finds the fix, and implements it in the codebase.
5. At some point in time, the fix is deployed.
6. The user is notified (well... maybe).

Typical lead time in the process above for anything other than showstoppers is not really minutes or hours. Think weeks or months. Why this kind of process exists in the first place is to filter the incoming requests to manage the team workload; a bit like in our public healthcare, you need to talk to a nurse (sometimes several) before you can see the doctor.

Wouldn't it be cool if steps 1-5 could be done in minutes, or half an hour (or to get directly to see the doctor)? Having step 6 still behind manual control for QC purposes might be needed, but in clear cases, when for instance our *ITIL triage agent* is able to verify from other sources that a fix is indeed needed, go all the way.

So coming to your nearest software support center, the fully automated repair workshop with the the following service menu:



Examples above about a software factory autonomously fixing products just based on consumer feedback may still sound a bit crazy, but not the 100% science fiction that it would've been just a couple of years ago.

Technical feedback loop

Another class of signals a software team can and should respond to is the technical stuff. Error rates, performance regressions, resource usage, and other technical signals are all measurements of how our software is performing in production. By monitoring these signals and responding to them

quickly, we can ensure that our software continues to meet the needs of our users and remains reliable and performant.

Automatic error detection has been in place in several areas for some time now, but the idea of having an *AI triage agent* that can not only detect errors but also suggest fixes and even implement them is a new one. That is, if we go beyond naive rebooting or increasing capacity on demand.

This category of errors is harder to fix automatically beyond the obvious 'reboot if mem is >90%'. AI models have been widely applied, way beyond this current *generative AI* wave, e.g. to detect anomalies in logs or network traffic.

Maintenance feedback loop

The maintenance engineers who work in your factory or the ones working in the field taking care of your equipment are a great source of feedback, too. They have a unique perspective on your software project, and typically have a lot of experience with the software and its quirks.

In the real IT world this group are the people running your tier 2 support, check up on your CI/CD systems, handle your user accounts etc, but are perhaps not directly involved in the development of the software. Much of the feedback is hence about small improvements to the software, or to the process, that can make their life easier. For instance, they might report that a certain error is happening more often than it should, or that a certain feature is not working as expected. They might also report that a certain process is taking too long.

If we'd apply the requests from the designated experts as first-class citizens who are to be trusted, perhaps a similar agentic feedback response factory as we suggested to handle the end-user feedback loop could be implemented to handle the maintenance feedback loop as well. Or, perhaps the maintenance engineers could act as curators of the technical signals, and suggest changes to the product or even the software itself, and instead of going through the usual process, the *AI triage agent* could take care of it directly.

Code quality and architecture compliance

There's a feedback loop that sits between the product-focused loops above and the factory-focused loops below: is the code we're shipping actually *good*, or, at least according to our standards? Or is it yet another very different implementation that deviates from other similar features we might have?

Who knew developers have their personal handwriting?

This matters more in the agentic world than it ever did before. When humans write code slowly, architecture drift is gradual and usually caught in review. When agents generate code fast, entire

subsystems can diverge from the intended architecture between Monday and Friday. The volume amplifies every quality problem.

The signals here are familiar, and some of them are possible to auto-detect to some degree. They might be for instance code smells, duplication, complexity metrics, dependency violations, or test coverage gaps. We need to address these effectively. Otherwise we'll risk our app becoming a big ball of mud spaghetti which will be hard also for agents to maintain later.

First line of defence is to inject these compliance checks after each agentic handoff; from planning all the way to review. What I'm after here really is how to improve these handoff points, their instructions and the architecture documentation to prevent these issues from happening in the first place.



Architecture compliance isn't just a gate at review time. When agents drift from architectural intent, the fix isn't more review. It's better constraints upstream: clearer conventions, tighter architecture decision records, and explicit boundaries in the agent's context.

Drift Detected — What Gets Refined?

Each deviation points to an upstream artifact that needs updating

DRIFT SIGNAL

Inconsistent patterns

e.g. duplicated logic across modules

Wrong architecture

e.g. direct DB calls from handlers

Style violations

e.g. naming conventions ignored

Bad generation patterns

e.g. missing error boundaries in agent output

Weak review coverage

e.g. tests coupling to internals

UPDATE ARTIFACT

Project instructions

add explicit pattern rules

Decision records

document chosen patterns with rationale

Conventions & linters

encode as automated checks

Agent specs & prompts

constrain what agents generate

Gate criteria

add checks that catch it pre-merge

Fix the source, not the symptom.

I've listed some simple examples above what kind of deviations you are likely to encounter, and what might be the corrective actions to take. The feedback loop here is about improving the project-level instructions, the architecture documentation, and perhaps even the agent recipes to make sure that the next time we run the factory, we get better output.

Per each run, or periodically, collect a list of deviations and verify what kind of adjustments you need for your project-level instructions, architecture documentation, or the agent recipes.



The feedback loops need circuit breakers as well. As tempting as a fully self-correcting software factory might sound, in the end **the signal-to-noise ratio of this transformation** might not be very high. Essentially we're using AI to improve itself, and that has its limits. For now, collecting periodic performance data, even a 'self diagnostics' report from the factory, and then reviewing it together with the team to decide on the improvements to make is perhaps the best way to go.

Developer and agentic feedback loops

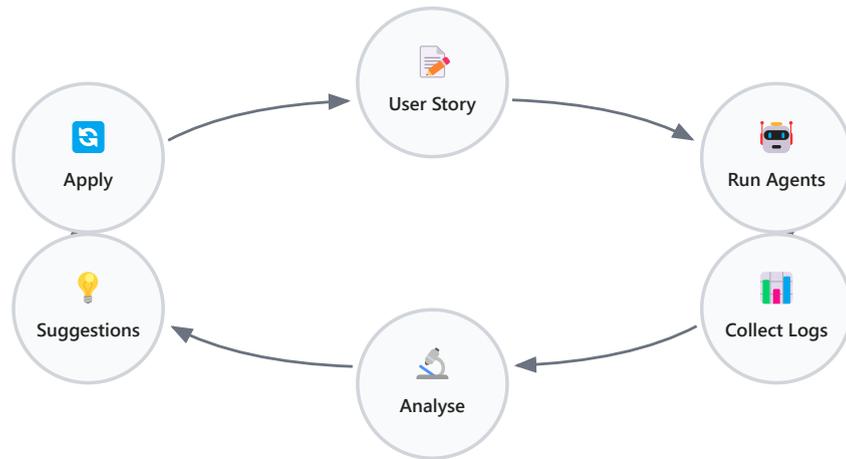
Finally, we have the feedback loop that is perhaps the most important one for the software factory itself: the feedback from the developers who are using the factory to build their software. (Yes, there will be people involved in this for some time, at least to some extent.)

Think issues like wandering agents taking hours to do simple stuff, too much to fix and iterate, unpredictable output or behavior. In the current world these things are not easy to fix. I have made some very limited attempts to have this kind of feedback loop in place, albeit human-triggered, to try to improve the agents and their context material, such as the documentation, after each run to act better next time.

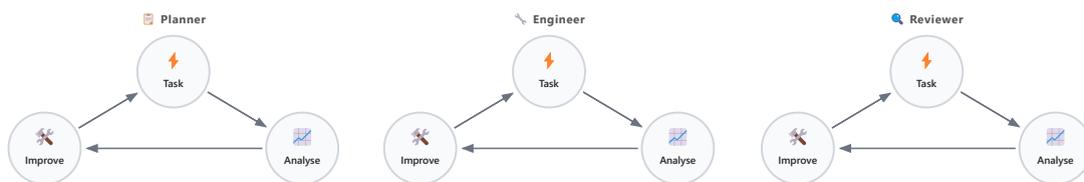
I've developed a naive, manual loop with two example tasks to perform. I collected all the logs, including task calls, spawns and user adjustments, and then used that as input for my best-practices-agent to suggest improvements on agent recipes, tool usage, wandering, separation of concerns, and so on. So I basically used AI to improve AI, and after two rounds of corrections our tool chain performed much better.

Factory Improvement Loops

MANUAL / BATCH IMPROVEMENT



PER-AGENT SELF-CORRECTION



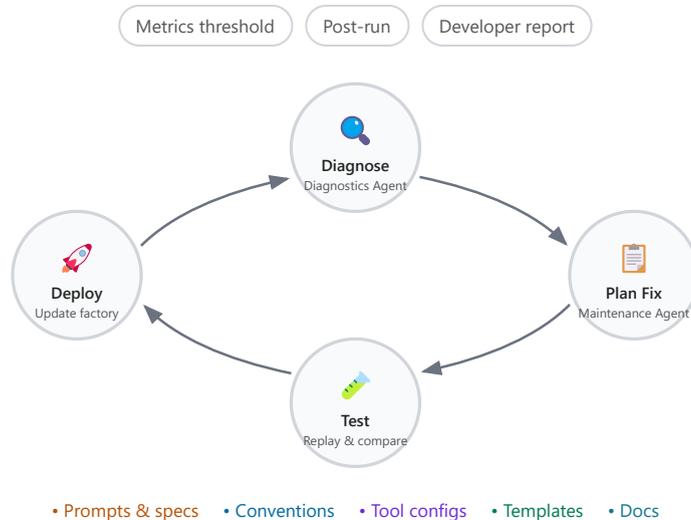
This loop was run manually with prepared examples that represented the typical tasks we had.

Similar feedback-collection could also be triggered by the factory itself. For instance, if an agent takes too long to do a simple task, or if it produces output that is not useful, the factory could automatically trigger a feedback loop to try to improve the agent's performance. This could be done by analyzing the agent's behavior and suggesting changes to its prompts, context material, or even its architecture.

Or the developer might tell an agent, 'this tool call keeps failing, why?' Or, 'why do my custom instructions or document keep being ignored?'

Factory Self-Improvement Loop

Agents that fix the agents — maintaining the machinery, not the product



We are in the early stages with our self-diagnosis and healing. This could be taken to the next level though, as illustrated above. Even now, it is possible to gather statistics about your runs, like duration, token usage, number of tool calls, and so on. Let's define boundaries for this, add another 'Diagnostics Agent' to watch the agent workflows to suggest improvements and optimizations, and apply them automatically.

- ✓ Depending on the Agent Harness (Codex, Claude Code, Open Code, Copilot) you use, the **detailed agent logs are an absolute goldmine** and a deep-dive inside the *agentic mind*. Especially the Copilot logs are very detailed, and rather educational especially if you wonder where did all that context window go, and what really was in it. Might be a good idea to store them periodically to catch drift, token usage, runtime and also number of retries and such to keep your machine working smoothly (or, as smoothly as possible).

Feature feedback loop

Finally, let's discuss how to assess the quality of the specifications we are feeding to the factory. Were they clear enough? Did they correctly capture your intent, all the edge cases? Did they have the right level of detail? Did they include the right context material?

There are a few approaches I could think of that could be used to develop this further.

- ? Did we miss any features? Despite asking for instance a specific navigation path, it's not there.
- ? Did our implementation do something close but not exactly what we wanted?
- ? Did we get unwanted features? A menu not needed, an entire page not requested, a new command-line switch for your tool that you didn't ask for?
- ? Were too many E2E tests generated? What was the 'Farley score' of the tests?
- ? Were there lots of user-reported bugs during the testing?
- ? Were the architecture, patterns and other conventions followed?

So roughly speaking there are two different kinds of mistakes:

1. Ignoring recurring patterns and practices, i.e. deviating from project-level instructions
2. Deviating from the specific instructions for the task you're trying to get done

Probably both lead to different corrective actions. The first one is more about improving the project-level instructions, the architecture documentation, and perhaps even the agent recipes to make sure that the next time we run the factory, we get better output. The second one is more about improving the specific instructions for the task at hand, and perhaps also improving our review process to catch these kinds of mistakes before they make it to production. To conclude, I've thrown in a couple of ideas worth a shot if (when) you encounter these kinds of defects in your factory.

Fixing deviations from project standards and practices: semi-manual feedback loop

How to create a feedback loop for the first one is perhaps more straightforward. You can have an agent that continuously monitors the codebase for patterns and practices. That's already baked in the flow, but as it is with gen AI, they can be missed or violated. Key would be to detect *why* they were violated, and to suggest improvements to the project-level instructions or the architecture documentation.

A naive solution for this kind of deviation could be something like:

► Example

- > **You:** "Hey factory, I noticed that the engineering agent keeps generating code with deeply nested conditionals, which violates our coding standards. Can you analyze why this is happening and suggest improvements to the agent's constraints or the architecture documentation to prevent this in the future?"

- ◆ **Factory:** "Sure! I've analyzed the recent code generations and found that the agent is defaulting to a procedural style when it encounters complex logic. This is likely because our current architecture documentation doesn't provide clear guidance on handling such scenarios. I suggest we update our architecture decision records to include specific conventions for managing complex logic, such as using design patterns like Strategy or State. Additionally, we can enhance the agent's constraints to prioritize these patterns when generating code with complex logic. I'll go ahead and implement these changes to improve the agent's performance in this area."
- › **You:** Ok, propose the changes, and let's review them before you implement them.
- ◆ **Factory:** "Here are the proposed changes to the architecture documentation and the agent's constraints. [Provides a summary of the changes]. Please review and let me know if you have any feedback or if you'd like me to proceed with implementing these changes."
- › **You:** "The proposed changes look good. Proceed."

For now, I think this particular 'drift' should be solved case-by-case as there's a far bigger risk of breaking things by giving agents too much power on the general topics.

Fixing deviations from task-specific instructions

The second class of mistakes is related to the specific task at hand.

By nature (depending on the scale of your mistake of course) these have a smaller blast radius than altering common practices. It's only one feature going bad, right? Well, this depends -- you might end up bricking some common service, template, or script and cause havoc by breaking 100 E2E tests.

A common mistake is saying one thing at the beginning of your specification ('this box should be openable from the top'), but something opposite at the end ('box should not be openable at all'). The longer your specification gets, the more difficult, even for AI, detecting this kind of inconsistency becomes.

A long-time customer of mine stated that the core rule of the software business is that the customer pays for the bugs and the supplier pays for fixing 'em.

So we're talking here about **errors in specification**. Back in the old days it might have been picked up by the reviewer, perhaps not.

Fixing this class of errors should be central to your Software Factory, as you don't want to produce invalid products. So, instead of relying on ralph-looping to get your way through, invest in a good, standard format for your specification (something easily understandable for both humans and

machines), and make sure to have a good process in place to review and improve them based on the feedback you get from the factory.

! **Becoming a good 'AI whisperer' takes time and practice. This is why the 'prompt engineering' never really worked: a small difference in prompt, a different or updated model might give a different answer next time you run.**

21

Software Engineering's Third Golden Age

Historical context and where this is heading

G rady Booch, known as one of the founders of UML and a lifelong voice in software engineering, frames the current moment as the "third golden age of software engineering."

➔ [Grady Booch on the Third Golden Age of Software Engineering](#)

According to Booch, the first golden age was about algorithms (1940s to 1970s), the second was about object-oriented abstractions (1970s to the 2000s), and the third golden age is about systems. It started with the rise of abstraction from individual components to whole libraries, platforms, and packages, not with the recent AI boom. Though AI fits into this, as it helps create even more complex systems with less effort than before.

Three Golden Ages of Software Engineering



While I could think of alternative abstractions and viewpoints to augment Booch's framing, I find it compelling and useful. It provides a historical context for the current moment, and it helps us understand the broader forces at play. The rise of systems and AI is not just a technological shift but it's a big change in how we think about software engineering itself. I

I would not consider object-oriented a golden age in its own right, but Grady has a point there: it began with punch cards, then we got VB and Java, and now we don't need to code at all!

might add that it might turn out to be way more fundamental than object-oriented programming ever was.

Existential crises - remember the Y2K?

Booch recalls that when compilers and higher-level languages emerged, developers feared obsolescence then, too. Well, our profession evolved. All in all computer science and programming are an astonishingly young field: "The term 'digital' was not coined until the late 40s, the term 'software' was not done until the 50s." Some of the existential dread about AI is happening in an industry that's barely 70 years old, out of which the first 40 were rather small-scale.

There's been so many 'crises' or 'disruptions' already in my 30 years in the industry. Internet, mobile, DevOps, Agile, Y2K, Blockchain, IoT, low code/no code, SaaS, offshoring, and now AI. Each one was hyped a lot, and yet here we are, still building software, still needing engineers.

In an event long time ago, a presenter (needless to say, rather 'high up') was confident that instead of writing code, we shall start building 'components' to a platform. Yes, *the* platform was the he was most familiar with, let's say it was a *big name*. There wasn't anything that could not be done with it. So his conclusion was that custom code/proprietary systems are dead (at least for us), and we should just start selling those 'reusable components' to customers waiting in the lobby with deep pockets. Yeah I might be cutting corners here, but that was pretty much the gist of it, and I'm quite sure many of you have heard similar claims in our industry.

As you might have already guessed, none of those components ever saw daylight. The presenter moved on after a couple of years after his 'components are the future' speech, and we continued coding as nothing happened. And here I am, after all these years, making a living creating proprietary software as many of my colleagues listening to the same talks still do.

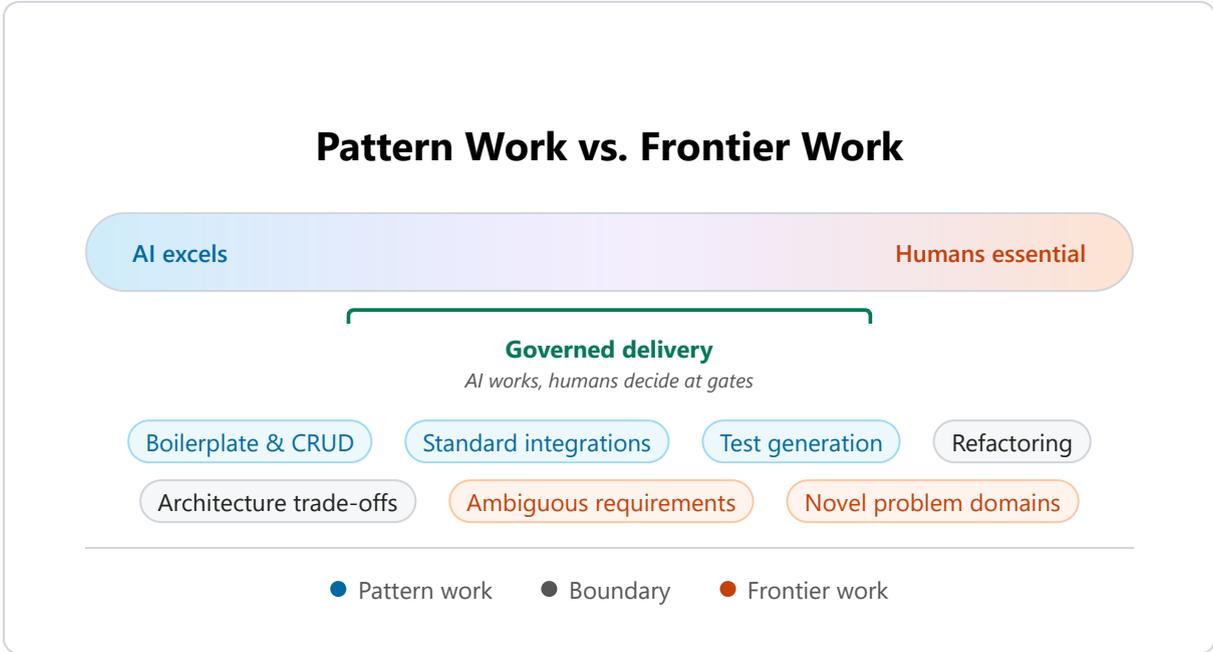
How I see it is that AI coding tools represent another rise in abstraction, not the end of engineering. You could think of it just as we moved from assembly to C or Pascal or whatever, even down to object-oriented programming. AI assistants are "akin to what was happening with compilers in those days."

Pattern work vs. frontier work

Every problem in software sits somewhere on a spectrum. At one end we have **known generic-use patterns**, such as standard CRUD operations, standard (well...) integrations, and well-understood algorithms. At the other end of this range we have the larger, **frontier problems**, such as novel architectures, domain-specific logic, the decisions about *what* to build and *why*.

Modern LLMs have been trained on the entirety of the public Internet: every Stack Overflow answer, every GitHub repository, every tutorial and blog post. That's an enormous library of patterns. It's why AI agents can be remarkably effective at the pattern end of the spectrum, and why they struggle at the frontier end. They can recombine what's been done before; they cannot reason about what hasn't.

This is precisely where the step size principle from [Chapter 5](#) meets reality. Governed decomposition works because it breaks frontier-scale problems into pattern-sized tasks, moving work from the part of the spectrum where AI struggles to the part where it excels.



But there's a concern worth noting. As AI-generated code floods the training pipeline, future models may increasingly train on their own output. Research published in Nature suggests this recursive loop degrades model quality over time — a phenomenon the authors call model collapse.

[↗ AI models collapse when trained on recursively generated data](#)

If the pattern library itself starts degrading, the boundary between pattern work and frontier work shifts, and the judgment we've discussed throughout this book becomes even more critical.

Deep foundations matter more

Our profession is moving at an incomprehensible pace towards automation. The people who will thrive in this environment are the ones with deep foundations, the ones who understand *why* systems work, not just how to use the tools built on top of them.

This is one part I'm genuinely worried about. When AI handles the routine implementation, there's less incentive to learn the basics. Programming from first principles, SQL, computer architecture, software design. But those are exactly the skills you need to tell whether an AI's output is correct, and to fix it when it's not. If we lose the ability to understand the systems we build, we can't govern them either. How do you validate something you don't understand?

Another concern is that we will become very dependent on this new technology. Model availability, training data quality, the geopolitical and economic forces behind these systems. That's not a reason to reject AI, but it is a reason to keep our own skills sharp. If you don't see it necessary just for the sake of **engineering craftsmanship**, then consider this for the sake of our own agency and autonomy.

I came into this field knowing how computers work from the silicon up. Semiconductors, networking, operating systems. I'd still argue every engineer should start with C and a plain text editor, and try to make something work from scratch. That experience builds the mental model you'll need when the AI gives you something that looks right but isn't.

What's next

After this book? I'll first do something not involving computers at all.

Professionally, I believe there's a lot to be learned about enhancing the reliability of this new technology, and how to govern it. I remain skeptical about unopinionated generic tools and one-size-fits-all models. For some years to come you'll still need to build your own factories, at least upwards from the foundation. I've found that strangely interesting and fun.

@ The roguelike was my proof of concept. A solo hobby project with no deadline and no client, where the governed flow still made the result better than winging it would have. Not every lesson from it scales to enterprise delivery, but the core ones do: plan before you build, verify before you move on, and stay in the decisions that matter.

Closing

This book is a collection of experiences, lessons, and opinions on how to build software with AI agents. But if there's a single thread running through all of it, it's this: the technology works when you govern it, and it fails in predictable ways when you don't.

The compound probability problem hasn't gone away. Each step in an AI pipeline still compounds uncertainty, and no model improvement has changed that fundamental math. What has changed is that we now know how to manage it: smaller steps, clear specifications, enforced gates, and humans at the checkpoints that matter. None of this is revolutionary. It's engineering discipline applied to a new kind of tool.

The people I've seen succeed with AI-assisted development share a few traits. They invest more time in planning than feels comfortable. They resist the temptation to let the AI run unsupervised just because it can. They treat specifications as the product and code as the derivative. And they stay curious about what's not working, not just what is.

I do agree with Grady Booch that we are in a golden age. This is, hands down, the most exciting time to be a software engineer during my career, which started in 1997. I don't believe we can put the genie back in the bottle. But we, as a profession, must learn to govern these systems that will produce most of the new program code going forward. We need to shift our focus to the *what* and the *why*. In a way, that might just be a good thing.

! AI-assisted development is here to stay. The question was never whether to adopt it, but how to make it reliable. Govern the process, stay involved in the decisions that matter, and build on foundations you actually understand.