The State of Vibe Coding

The History, Art, and Concepts of Vibe Coding

Hyunwoo Park and contributors · SNU Special Topics in Data Science: Vibe Coding

2026-09-14

Table of contents

Preface

This book is an attempt to gather thoughts about vibe coding in one place. Coined in Andrej Karpathy's tweet in February 2025, the term now serves as a lens not just for a particular coding style but for an entire way of working and studying with AI. If prompt engineering was its prehistory, loop engineering is its next chapter. This book covers that whole arc.

The nature of this book

It is a living book. In the Seoul National University Special Topics in Data Science course "Vibe Coding," the instructor and students write it together on GitHub. A new edition comes out each semester, and it is revised frequently in between. It is not a finished book but a snapshot updated every year, literally "the state of" vibe coding.

It comes in two formats. A web edition and a typeset PDF edition are built together from the same manuscript.

It comes in two languages. The Korean edition is the primary original, and the English edition is machine-translated by AI. However, for chapters whose authors are more comfortable writing in English, English is the original and Korean is the translation. The original language of each chapter is noted in the metadata at the top of the chapter.

What it covers

- **History**: the evolution of methodological discourse from prompt engineering to vibe coding, and on to agentic coding and loop engineering
- **Techniques**: the opportunities vibe coding opens up, best practices, and hands-on practice for what can be made, from writing and code to web applications, images, and video
- **Principles and limits**: why this approach works, and where it breaks down, including the problem of AI-touched codebases turning into black boxes
- **Outlook**: how AI is changing the way we study and work
- **Concept dictionary**: a dictionary of concepts being born and evolving around vibe coding. It keeps growing as an appendix

How to contribute

This book itself is both a product of the course and a hands-on workshop. See the repository's CONTRIBUTING.md for how to contribute. How the course is run is documented in the README and the syllabus.

> ℹ Current edition
>
> The 2026 edition (in progress). Writing is proceeding alongside the second-semester 2026 course.

Part I · History of Vibe Coding

1 Prehistory: A History of Humans Speaking to Machines

Vibe coding did not appear suddenly one day. The dream of "programming in human language" is as old as the history of computing itself, and it has returned

again and again under different names. This chapter begins with that history of repetition and arrives at prompt engineering, the direct prehistory of vibe coding.

1.1 The Old Dream of Natural Language Programming

> ⚠ Drafting Status
>
> Pre-draft. Check ownership and status in TOPICS.md at the repository root.

Brief. The idea that "writing code in natural language removes the barrier to entry for programming" did not start with vibe coding. From the "English-like syntax" that COBOL championed, to fourth-generation languages (4GLs), to CASE (Computer-Aided Software Engineering) tools, a similar promise has recurred over decades, and each time it was never fully realized. This section should trace the history of that recurrence and lay out specifically what each attempt promised and where it fell short. The goal of this section is to establish grounds for judging whether vibe coding should be seen as the latest iteration of this history, or as a qualitatively different break from it. This is the section that covers the most distant origin in all of Part 1.

Seed Questions

- What did COBOL's designers actually aim for with "code that reads like English," and to what extent was that goal achieved?
- What problem did 4GLs claim to solve when they emerged, and what trajectory did their actual adoption follow?
- What specifically were the promises of CASE tools and the failure cases of the 1990s?
- Is there a common cause behind the repeated failures of natural language programming, or did each attempt fail for a different reason?
- If there is a basis for vibe coding to claim "this time is different," what is it, or is it just a repetition of the same pattern?

Research Pointers

- Check COBOL's original design documents (CODASYL-related materials) and records of Grace Hopper's related remarks in primary sources.
- Cross-check the accounts in software engineering textbooks (e.g., Pressman, Sommerville) that cover the history of 4GLs and CASE tools.
- Search for academic reviews or retrospective pieces covering the repeated failures of natural language programming.

1.2 End-User Programming: The Spreadsheet Precedent

> ⚠ Writing Status
>
> Not yet drafted. Check ownership and status in TOPICS.md at the repository root.

Brief. There was already an era when non-programmers did programming. Writing cell formulas in a spreadsheet is, formally, programming, and it is the most widely adopted case of end-user programming for decades. This section should analyze what made spreadsheets successful (verifying candidates such as immediate feedback, visual representation, and a low barrier to entry) and draw out the implications for vibe coding. At the same time, it should also address the counterexample that spreadsheet errors have repeatedly caused major incidents in practice. In Part 1, this section serves as the counterpart that approaches the theme of "programming by non-experts" from the angle of tool form rather than programming language.

Seed Questions

- How has the term end-user programming been defined in academia (HCI, programming language research)?
- Why were spreadsheets adopted far more widely than other natural-language or low-code attempts?
- What actual incidents have resulted from spreadsheet errors (in finance, academia, etc.), and what do they suggest?
- In what ways is the spreadsheet's immediate execution feedback structure similar to and different from vibe coding's execution-verification loop?
- Which success factors of spreadsheets have low-code/no-code platforms inherited, and which have they missed?

Research Pointers

- Search the end-user programming research literature (the HCI/PL crossover field, e.g., research by Burnett and others).
- Check collections of spreadsheet error cases (such as the incident archive of the European Spreadsheet Risks Interest Group).
- Verify exact years and figures in computing history sources covering the adoption history of VisiCalc, Lotus 1-2-3, and Excel.

1.3 The Short Heyday of Prompt Engineering

> ⚠ Writing Status
>
> Pre-draft. Check ownership and status in TOPICS.md at the repository root.

Brief. In the stage just before vibe coding, there was a heyday of prompt engineering that looks brief in hindsight. This piece should cover the period when "what to say and how to say it" to a large language model was treated as a standalone skill, the process by which techniques like few-shot and chain-of-thought were discovered and formalized, and the process by which the job title "prompt engineer" appeared in the job market and then quickly faded. The key question is what limitation of this technique-centered era brought about the next stage, vibe coding, which centers on iterative dialogue and execution verification. In Part 1, this piece serves as the final prehistory installment right before the transition to Chapter 2 (Birth).

Seed questions

- When and in which community did the term "prompt engineering" first come into use?
- What problem was each representative technique, such as few-shot, chain-of-thought, and role assignment, developed to solve?
- Where, and why, did the boundary between "the skill of writing a good prompt" and "the skill of building something through iterative dialogue with a model" start to break down?
- When did prompt engineer job postings peak, and when did they start to decline?
- Can Karpathy's Software 2.0 (2017) discussion be read as having foreseen the prompt engineering era?

Research pointers

- Check Anthropic's and OpenAI's official prompt engineering guide documents as primary sources
- Check the publication timing and core arguments of early academic papers on chain-of-thought (Wei et al. and others)
- Collect news articles covering the rise and disappearance of prompt engineer job postings
- Reread Karpathy, "Software 2.0" (2017) for how it connects to the discourse of this period

2 The Birth of Vibe Coding

In February 2025, a single tweet from Andrej Karpathy gave a name to a way of working that had no name until then. This chapter dissects that moment: what the original text said and did not say, why that particular moment, and how the meaning drifted as the word spread.

2.1 Karpathy's Tweet and the Conditions of That Moment

⚠ Writing Status

Pre-draft. Check ownership and status in TOPICS.md at the repository root.

Brief. In February 2025, a single tweet by Andrej Karpathy gave rise to the term that would be cited more than any other in the software industry over the following year. This section dissects that original text: exactly what it defined and what it did not, how the technical conditions at the moment the tweet was written (the model generation, the maturity of agentic coding tools, the mood of the developer community) contributed to the spread of this definition, and why that particular moment. This is the section that serves as the starting point for the whole of Chapter 2.

> There's a new kind of coding I call "vibe coding", where you fully give in to the vibes, embrace exponentials, and forget that the code even exists.
>
> — Andrej Karpathy, X, February 2025

Seed Questions

- What scope did Karpathy's original text actually define, and how does it differ from the meaning that later became widespread?
- What coding agents and tools were available at the time the tweet was posted, and what role did their maturity play in the adoption of this concept?
- A year later, in a retrospective tweet, Karpathy called this tweet "a thought he tossed out after it came to him in the shower." How does this self-assessment diverge from its actual impact?
- In the early days as the tweet spread, which reactions (welcome, cynicism, refutation) predominated?

Research Pointers

- Collect the full text of the original tweet (https://x.com/karpathy/status/1886192184808149383) and the one-year retrospective tweet (https://x.com/karpathy/status/2019137879310836075), along with major quote-tweet reactions.
- Check the release and update history of major coding agent tools around the time the tweet was posted.
- Check the Wikipedia overview (https://en.wikipedia.org/wiki/Vibe_coding) for descriptions related to early adoption.

2.2 Spread of the Term and Drift of Its Meaning

⚠ Writing Status

Pre-draft. Check ownership and status in TOPICS.md at the repository root.

Brief. A term that started from a single tweet came to be used, within a few months, in a much broader sense than its original definition. This section covers that spread and the drift of its meaning. It should cover why Simon Willison drew a line by saying "not all AI-assisted programming is vibe coding," how the narrow

sense (fully delegating without reading the code) and the broad sense (coding in general using AI tools) circulated at the same time, and how the Collins Dictionary, by selecting it as its Word of the Year, gave this spread official sanction. If § 2.1 in Chapter 2 covers the moment of the utterance, this section covers the process of circulation that followed.

Seed questions

- What was the specific criterion for the line Simon Willison drew? What examples did he use to distinguish between "real vibe coding" and "AI-assisted programming"?
- What kind of confusion (debates like "is this vibe coding?") did the coexistence of the narrow and broad senses cause in practice?
- How did the Collins Dictionary's citation for its selection define the term? Is that definition closer to the original tweet or to Willison's distinction?
- When comparing snapshots across time (right after the tweet, months later, at the time of the dictionary entry), what pattern appears in the drift of meaning?

Research pointers

- Read the full text of Simon Willison, "Not all AI-assisted programming is vibe coding" (2025-03-19)
- Compare the definition wording in the Collins Word of the Year 2025 announcement (https://www.collinsdictionary.com/us/woty) with the news coverage (https://www.cnn.com/2025/11/06/tech/vibe-coding-collins-word-year-scli-intl)
- Check whether the change in the definition can be traced through the edit history of the Wikipedia overview (https://en.wikipedia.org/wiki/Vibe_coding)

2.3 Genealogy of Tools: From Autocomplete to Agents

⚠ Writing Status

Pre-draft. Check assignment and status in TOPICS.md at the repository root.

Brief. This section deals with the genealogy of tools, not concepts. It should trace the flow starting from GitHub Copilot's autocomplete-style coding assistance, released in 2021, through chat-interface-based coding tools, to agentic CLI and IDE tools. The core claim is that the form of the tool (autocomplete versus conversation versus autonomous execution) has defined the coding methodology of its era. We need to verify the causal claim that the methodology of vibe coding would not have emerged without the tool forms that made it possible. Within Chapter 2, this section complements the conceptual history (§ 2.1, § 2.2) with a history of tools.

Seed Questions

- Why did GitHub Copilot (released in preview in 2021) choose the autocomplete form, and which later tools adopted that form as their default?

- The transition from autocomplete-style tools to chat-based coding assistance (code explanation, generation requests) occurred with which tools, and at what point in time?
- What technical prerequisites did the transition from chat-based to agentic tools (file system access, command execution, autonomous loops) require?
- How did the developer community react (welcome, resistance) at each transition period between tool forms?
- Which led which, tool form or methodology, or were they mutually determining?

Research Pointers

- Check GitHub Copilot's public announcement (2021; confirm the exact date and distinguish preview from general availability) and its subsequent major update history.
- Compile the release chronology of representative agentic coding CLI and IDE tools (Cursor, Windsurf, Claude Code, Devin, etc.).
- Check the discussion of tool form and autonomy in Anthropic's "Building effective agents" (December 2024).

3 Evolution of the Discourse

As soon as the term vibe coding took hold, a flood of concepts emerged to refine or move beyond it: CHOP, context engineering, spec-driven development, loop engineering, vibe engineering. This chapter traces the genealogy of these concepts and tracks what bottleneck each conceptual shift was responding to. The final section treats this marketplace of concepts itself as an object of analysis.

3.1 CHOP and Agentic Coding: A Spectrum of Autonomy

> ⚠ Writing Status
>
> Not yet drafted. Check TOPICS.md at the repository root for ownership and status.

Brief. This section starts from the concept of CHOP (chat-oriented programming), which Steve Yegge used before the term vibe coding spread, and moves on to the agent autonomy spectrum (how far humans are involved and from what point agents move autonomously) that was formulated afterward. The key question is how CHOP, vibe coding, and agentic coding are distinguished from and overlap with each other. Chapter 3 covers the follow-up concepts that poured out after the term vibe coding took hold, and this section covers the earliest of these threads.

Seed Questions

- When and in what context did Steve Yegge first propose CHOP, and what is its chronological relationship to vibe coding?
- How do CHOP and vibe coding differ by definition, and do practitioners use the two terms distinctly or interchangeably?

- How does the term “agentic coding” relate to CHOP and vibe coding respectively?
- Who organized the attempts to divide agent autonomy into stages (from autocomplete to autonomous execution and autonomous verification), and by what criteria?
- How did the book co-written by Gene Kim and Steve Yegge synthesize these concepts?

Research Pointers

- Check Steve Yegge’s remarks on CHOP in the O’Reilly podcast interview
- Check the exact publication date and core arguments of the book “Vibe Coding,” co-written by Gene Kim and Steve Yegge
- Find and compare practitioner blog posts that address stages of agent autonomy (cases that present a level table)

3.2 Context Engineering

> ⚠ Writing Status
>
> Pre-draft. Check ownership and status in TOPICS.md at the repository root.

Brief. This section covers the naming and formalization of the concept of context engineering. Shopify CEO Tobi Lütke coined the term in a June 2025 tweet, and Karpathy lent weight to its formalization by endorsing it. The core claim is that the bottleneck has shifted from the skill of writing a single good prompt to the skill of designing the entire construction of the context window — what the agent sees and doesn’t see on each turn. In Chapter 3, this section covers the concept that directly succeeds prompt engineering (Part 1).

Seed Questions

- In exactly which tweet did Tobi Lütke propose this term, and what problem awareness did it arise from?
- In what way did Karpathy agree with this naming, and what role did his endorsement play in the term’s spread?
- What is the substantive difference between “prompt engineering” and “context engineering,” and where do they overlap?
- How did Simon Willison and Phil Schmid each summarize this concept, and is there a difference in emphasis between the two summaries?
- What development practices has the concept of context engineering actually changed (documents like CLAUDE.md and AGENTS.md, context window budget management, etc.)?

Research Pointers

- Check Tobi Lütke’s naming tweet (2025-06-19) and Karpathy’s endorsement tweet
- Read Simon Willison’s summary post (2025-06-27)

- Read Phil Schmid's summary post

3.3 Spec-Driven Development

> ⚠ Writing Status
>
> Pre-draft. Check ownership and status in TOPICS.md at the repository root.

Brief. This section treats spec-driven development as a reaction to vibe coding. Spec Kit, released by GitHub in September 2025, is a tool that explicitly walks through four stages: Spec, Plan, Tasks, and Implement. The core question is what tension exists between vibe coding's attitude of "forget that the code even exists" and spec-driven development's attitude of writing the specification first and prioritizing it over the code. It should also address whether this conflict is a replay of the waterfall versus agile debate (revisited in Part 4 § 10.1). In Chapter 3, this section covers the first systematic reaction to vibe coding.

Seed Questions

- What problem was each of the four stages specified by GitHub Spec Kit (Spec, Plan, Tasks, Implement) designed to address?
- Are there other tools or workflows that claim to be spec-driven development, and if so, how do their approaches differ?
- Is spec-driven development an attempt to replace vibe coding or to complement it, and which is closer to practitioners' actual usage patterns?
- What similarities and differences emerge when this conflict is overlaid with the waterfall versus agile debate (Royce 1970, Agile Manifesto 2001)?
- Is there evidence that the spec-first approach actually improved code quality or verifiability?

Research Pointers

- Read GitHub's official announcement, "Spec-driven development with AI" (2025-09)
- Contrast the primary sources Winston Royce, "Managing the Development of Large Software Systems" (1970) and the Agile Manifesto (2001)
- Investigate and compare similar workflows besides Spec Kit (plan-mode-type features from other vendors)

3.4 Loop Engineering and Harness Engineering

> ⚠ Writing Status
>
> Pre-draft. Check ownership and status in TOPICS.md at the repository root.

Brief. This section covers two successor concepts formalized in 2026: loop engineering and harness engineering. Loop engineering is a concept that Addy Osmani formalized and popularized around June 2026; it covers designs that make agents

iterate on their own, finding and verifying work by themselves. Harness engineering is a concept that emerged from the formulation "an agent is the sum of a model and a harness," emphasizing the design of the tools, prompts, and verification mechanisms that surround the model rather than the model itself. The key question is where these two concepts sit, alongside prompt engineering and context engineering, on the continuum of bottleneck shifts.

Seed Questions

- What is the core claim of the loop engineering concept as formalized by Addy Osmani, and what practical examples does he cite?
- How does the "loop" as a unit (a cycle in which the agent finds work, does it, verifies it, and remembers it on its own) differ from earlier conversation-turn-based interaction?
- Who first proposed the formulation "Agent = Model + Harness," and when, and what does this formulation emphasize?
- In practice, how is the term harness engineering distinguished from context engineering, or do the two overlap?
- Which of loop engineering and harness engineering refers to the broader concept, or are they different axes altogether?

Research Pointers

- Trace back to Osmani's original formalization via the adtmag report, "Loop Engineering Emerges as Developers Put AI Coding Agents on Repeat" (2026-07-01).
- Read Faros AI's "harness engineering" blog post.
- Check the genealogical relationship with Anthropic's "Building effective agents" (2024-12).

3.5 Vibe Engineering and Its Reactions

> ⚠ Writing Status
>
> Pre-draft. Check TOPICS.md at the repository root for owner and status.

Brief. This section covers the concept of vibe engineering proposed by Simon Willison and the attempts to redefine it that arose around it. Once vibe coding came to be widely understood as an attitude of "not reading the code and leaving everything to it," terms emerged in reaction, referring to an attitude that maintains speed while taking responsibility for the results. This section needs to verify exactly what distinguishes vibe engineering from vibe coding, and whether these attempts at redefinition are actually used distinctly in practice. In Chapter 3, this section, together with § 3.3 (spec-driven development), forms the axis of reaction against vibe coding.

Seed Questions

- What problem was Simon Willison responding to when he proposed vibe engineering, and exactly when did he propose it?
- Specifically, where is the boundary drawn between vibe engineering and vibe coding?
- Are there other writers or concepts that have attempted redefinitions along the lines of "accountable acceleration," and if so, how do they compare with Willison's definition?
- Are these attempts at redefinition actually used distinctly in practice, or are they all lumped together under the term vibe coding?
- How widely did the term vibe engineering itself spread afterward, or did it remain a minority discourse?

Research Pointers

- Simon Willison, "vibe engineering" - check the full original text and the exact publication date.
- Compare the continuity of argument with Willison's earlier piece "Not all AI-assisted programming is vibe coding" (2025-03-19).
- Find follow-up pieces that cite or rebut vibe engineering to gauge the extent of its spread.

3.6 Observing the Concept Market: How Is the Next Buzzword Made?

> ⚠ Drafting Status
>
> Pre-draft. Check ownership and status in TOPICS.md at the repository root.

Brief. This section serves as a meta-analysis of the entire Chapter 3. It takes as its object of analysis the pattern itself of birth, spread, and disappearance (or settling) of the terms that led from prompt engineering to vibe coding, agentic coding, context engineering, spec-driven development, loop engineering, harness engineering, and vibe engineering. The core hypothesis is that each conceptual replacement reflects a shift in the actual bottleneck (from a lack of model capability, to a lack of context management, to a lack of verification, to a lack of loop and system design). If this hypothesis holds, we can also try to predict which bottleneck the next buzzword will point to. As the final section of Part 1, it serves to summarize the entire conceptual history before moving on to Part 2 (Technology).

Seed Questions

- If we arrange the concepts covered in this chapter in chronological order, how long was each concept's lifespan (from its emergence until it gave way to the next concept)?
- At each point of conceptual replacement, what can we claim was the bottleneck that actually shifted, and what is the basis for that claim?

- Did these discourses actually change development practices, or were they closer to just putting a new name on existing practices?
- Is there a common pattern in the paths by which concepts spread (a single tweet, endorsement by an influential figure, media coverage, dictionary entry)?
- If this analysis is correct, in which direction can we predict the next bottleneck and the next buzzword will lie?

Research Pointers

- Organize into a table the exact emergence timing of each concept as established in the other sections of this chapter (§ 3.1- § 3.5), and compare them chronologically.
- Search for whether there is linguistic or meme-diffusion research that addresses the mechanism of terminology diffusion itself.
- Examine whether Meir Lehman's laws of software evolution (1970s) can also be analogized to the evolution of concepts themselves.

Part II · The Art of Vibe Coding

4 Opportunities and Potential

What vibe coding opens up is not "coding faster" but the range of who can build and what can be built. This chapter examines that opportunity in order: people (who gets access), things (what becomes newly possible), economics (how the cost structure changes), and the research frontier.

4.1 Which Doors Opened for Whom

> ⚠ Writing Status
>
> Not yet drafted. Check TOPICS.md at the repository root for owner and status.

Brief. The first thing vibe coding changed is the boundary of who can make software. This section's theme is how widely this door has opened for non-programmers, domain experts, solo founders, and researchers respectively, and what new barriers have been hidden behind that threshold. The narrative that "now anyone can build" comes largely from success stories, while failure cases and cases of giving up partway are relatively underdocumented. This section gathers both currents in balance to concretely depict who actually gained access to what. It serves as the introduction to Chapter 4 (Opportunities and Potential), with the following § 4.2-4.4 covering the specifics.

Seed Questions

- What cases are there of non-programmers completing software from start to finish with vibe coding, and where did they get stuck?
- What patterns appear in cases where domain experts (medicine, law, education, etc.) directly coded solutions to problems in their own field?

- Did vibe coding actually let solo founders shrink their team size, or did it just shift the problem to other bottlenecks (marketing, operations)?
- What evidence do skeptical rebuttals to the claim that “now anyone can build” cite?
- Record your own experience of building something as a case for this section. What background knowledge were you missing when you first tried vibe coding, and what actually filled that gap, and what did not?

Research Pointers

- Collect build reports from non-developers (blogs, communities, Product Hunt-type sites), covering both successes and failures
- Case interviews from solo founder and indie hacker communities (Indie Hackers, etc.)
- Interview students themselves and their peers about their first building experience and compile them as cases

4.2 Software for One, Disposable Software

> ⚠ Writing Status
>
> Pre-draft. Check the repository root’s TOPICS.md for owner and status.

Brief. When the marginal cost of building software falls, a new category emerges: software built and discarded for just one person, or for a single task. This section addresses whether this kind of personal, disposable tool is actually increasing, and if so, in what form it appears. The existing software industry has been designed around the premise of reuse and economies of scale; when that premise is shaken, the key question is what remains and what newly emerges. This is also an answer to what the people who came in through the door opened by § 4.1 are actually building.

Seed questions

- Where can we find the terminology and discussion around personal, disposable software?
- In which domains do actual examples of disposable software (scripts and tools used once and discarded) appear most often?
- When the marginal cost of custom software falls, how do the existing SaaS and general-purpose software markets respond?
- How much maintenance, security, and documentation does disposable software actually need? Is the claim that none is needed at all valid?
- Record a case study of a disposable or personal tool you made yourself. What was it built for, and what happened to it afterward (still in use, discarded, shared with others)?

Research pointers

- Search practitioner blogs and talks using keywords like “software for one,” “disposable software”
- Collect similar cases from communities where people share personal tools (e.g., Hacker News Show HN-type posts)
- Gather and organize personal tools made by students through surveys or interviews

4.3 The Economics of Prototyping

> ⚠ Writing Status
>
> Not yet drafted. Check ownership and status in TOPICS.md at the repository root.

Brief. When the distance between the moment an idea occurs and the moment a working demo can be shown shrinks, the rationale for the various organizational practices that used to fill that gap (planning documents, approval processes, mockup tools) starts to waver. This section addresses what practices the change in prototyping speed has actually altered on the ground in startups and enterprises, and what has remained unchanged. It should also distinguish whether only the speed has increased, or whether the nature of the prototype itself has changed. It extends the discussion of personal software in § 4.2 into organizational and startup contexts.

Seed Questions

- How much has the time to build a startup’s MVP (minimum viable product) actually shortened? What is the quantitative evidence?
- Has the internal approval process for prototypes within companies shortened as much as production speed has increased, or has the bottleneck simply moved elsewhere?
- What problems arise as the boundary between a “quick prototype” and a “prototype deployed directly to production” blurs?
- How has the role of the prototype changed in fundraising or internal persuasion processes?
- Actually measure the time it takes you to go from idea to a working demo and record it as a case study. What was the bottleneck (conception, building, or validation)?

Research Pointers

- Collect mentions of MVP build times from public materials by startup accelerators and VCs (verify figures against primary sources).
- News coverage of internal corporate hackathons and innovation lab cases.
- Record students’ own project build times and use them as comparative data.

4.4 Vibe Coding for Researchers

⚠ Drafting Status

Not yet drafted. Check ownership and status in TOPICS.md at the repository root.

Brief. For researchers, vibe coding operates under somewhat different conditions. Data pipelines, visualizations, and simulation code are mostly not written once and discarded, but must pass peer review and reproducibility checks. This section covers how vibe coding is actually used in research settings, and where the vibe coding practice of "if it works, it's fine" collides with the research requirement that results "must be reproducible." Since this book is a product of the Special Topics in Data Science course at Seoul National University, this section also touches on the actual working methods of the author group itself.

Seed Questions

- How are researchers using AI coding tools to write data preprocessing, visualization, and simulation code? Are there differences across fields?
- Do reproducibility requirements (code disclosure, fixed environments, fixed seeds) actually conflict with the fast-iteration practices of vibe coding, or do tools resolve this automatically?
- In statistics and econometrics code, what procedures are needed to verify AI-generated results differently?
- What constraints do journal and conference AI-use disclosure policies place on writing research code?
- Record, as a case study, your own experience of vibe coding one of a data pipeline, visualization, or simulation for your research or coursework. How did you verify the results?

Research Pointers

- Accounts from econometrics and statistics researchers on using AI coding tools (blogs, X threads)
- Academic discussions of reproducibility (literature on the reproducibility crisis and its reexamination in the AI era)
- Compile each student's own experience building research or assignment pipelines as case studies

5 Best Practices

The gap between doing vibe coding well and doing it poorly is large, and most of that gap can be learned. This chapter organizes the practices practitioners have built up since 2025 around principles that will outlast any particular tool: loop size, planning, context, verification, recovery, and knowing when to stop.

5.1 Start Small and Check Often

⚠ Drafting Status

Pre-draft. Check ownership and status in TOPICS.md at the repository root.

Brief. The most frequently repeated advice among vibe coding practitioners is not to ask for too much at once. The rule of thumb is that the smaller the unit of each request and the more often the result is checked, the higher the quality of the final output. This section covers why this principle holds, what size is appropriate, and how this principle coexists with the original definition of vibe coding as "leaving it to the AI and just looking at the result." It is the first principle of Chapter 5 (Best Practices) and serves as the premise for the following § 5.2 and § 5.4.

Seed Questions

- Are there practitioners or tool documentation that quantitatively specify an appropriate size for a "small step"? (e.g., number of files, lines changed, time spent)
- What is the trade-off between keeping the iteration loop short and delegating a long task wholesale to an agent and waiting (handing the agent a long task in one piece)?
- What specifically does "checking the result often" mean? Is reading the code a different principle from just looking at the execution result?
- When this principle fails, under what conditions do cases arise where quality still deteriorates even after breaking work into small steps?
- Record, as a case study, your own experience comparing vibe coding sessions where you took large steps versus small steps.

Research Pointers

- Recommended workflow descriptions from official documentation of tools such as Claude Code and Cursor
- Collect descriptions related to iteration loop size from practitioner blog posts of the "how to code with AI" genre
- If students have their own session logs, directly compare the relationship between step size and result quality

5.2 Plan First: Starting with Spec and Plan Documents

⚠ Writing Status

Pre-draft. Check the repository root's TOPICS.md for owner and status.

Brief. The original meaning of the word vibe leans toward going by feel without a plan, but the practice that has settled in the field moved in the opposite direction. Before writing code, you have the AI write a plan document or spec first, and only after a human reviews it does the work move to the execution stage. This section

covers how this plan-first workflow came to be established, what tool-level mechanisms like plan mode were trying to solve, and how this practice differs from or overlaps with the spec-driven development covered in § 3.3.

Seed questions

- When and in which tools did tool features that explicitly support plan-first workflows (plan mode, plan approval steps, etc.) first appear?
- What is the appropriate level of detail for a plan document? What is lost if it is too detailed, and what goes wrong if it is too sparse?
- How do practitioners report their experiences of intervening to change direction at the plan stage?
- Are the spec-driven development in § 3.3 and the plan-first practice in this section the same thing, or do they differ in degree of formality?
- Record as a case study your own experience comparing having a plan document written first against having code written directly.

Research pointers

- Official documentation for Claude Code plan mode and other tools' planning and approval features
- Plan document templates or prompt examples published by practitioners
- Comparing outcome differences with and without a plan document in students' own projects

5.3 The Art of Context Management

> ⚠ Writing status
>
> Pre-draft. Check ownership and status in TOPICS.md at the repository root.

Brief. The recognition that what the agent knows and doesn't know at the start of a task heavily determines the outcome is the practical counterpart of the context engineering concept covered in § 3.2. This section covers the practical techniques of project instruction files like CLAUDE.md and AGENTS.md, how to allocate the context window's budget, and what to show and what to deliberately hide. It serves as a bridge between theory (§ 3.2) and practice (Chapter 5).

Seed questions

- What content do project instruction files like CLAUDE.md and AGENTS.md reportedly need to contain to be effective in practice?
- As context windows grow larger (e.g., high-capacity token models), does the need for context management techniques themselves diminish, or does it persist in a different form?
- What concrete advice do practitioners give on "what to hide"? (e.g., excluding unnecessary files, summarizing old conversations)

- How are cases reported where context management failures led to actual task failures?
- Record, as a case study, your own experience writing and improving a project instruction file (CLAUDE.md-type). What did you add that made the results better?

Research pointers

- Official documentation for project instruction files in Claude Code, Cursor, GitHub Copilot, etc.
- Practical guides on context engineering (cross-reference with § 3.2 research findings)
- Analyze this repository's (vibecoding) own CLAUDE.md as a case study

5.4 Creating Verifiable Units

> ⚠ Writing Status
>
> Pre-draft. Check TOPICS.md in the repository root for owner and status.

Brief. The phrase "it works" is a looser standard than it seems. It might mean that running it once produced no errors, or it might mean that all tests passed. This section covers the techniques used in vibe coding practice to narrow this loose standard: writing tests, directly checking execution results, and the practice of triangulation, cross-verifying the same result through multiple methods. It is the practical counterpart to the concept of verification asymmetry covered in § 8.3, and shows what techniques actually implement the theory that code is easy to verify.

Seed questions

- Is the practice of having AI write tests directly during a vibe coding session reliable? How is the risk of the tests themselves being poorly written addressed?
- "Confirming by running it" and "passing tests" are two standards with different verification strength. What combination do practitioners recommend?
- What are concrete examples of triangulation (reconfirming the same result through a different method) actually being used?
- How is the cost of splitting verification units too finely (slowdowns, excessive test code) discussed?
- Record the verification methods used in your own project in detail. What did you test, what did you only check visually, and why did you make those choices?

Research pointers

- Practical discussions on combining TDD with AI agents (can cross-reference § 10.4)
- Official documentation on automatic test generation and execution features by tool
- Compile students' own project verification procedures as case studies

Translated file written to `en/sections/05-5-reversibility.qmd`, matching the machine-translation marker format (source-sha256, model, date) used by `tools/translate` and the sibling `en/sections/05-2-plan-first.qmd`.

5.5 When to Stop and Read the Code Yourself

> ⚠ Writing Status
>
> Pre-draft. Check ownership and status in TOPICS.md at the repository root.

Brief. The difference between people who are good at vibe coding and those who are not mostly comes down to a sense of when to trust the AI and when to step in directly. This section works to turn that sense into as explicit a list of signals as possible. It covers the situations in which practitioners actually stop and read the code themselves, and what cases where that judgment was delayed and the problem grew larger show us. As the final section of Chapter 5, it closes out the practical principles carried through since § 5.1 with the question of when these principles break down.

Seed Questions

- What signals do practitioners commonly cite as requiring direct intervention? (security-related code, data deletion, payment logic, etc.)
- What patterns appear in real cases (post-mortems) where missing this signal caused a problem to grow larger?
- Does the standard for when to stop change depending on experience or domain knowledge? What is the difference in standards between beginners and experts?
- At the team level, are there attempts to replace this judgment with a process (review, gates) rather than leaving it to the individual?
- Record, as a case, a moment when you yourself actually stopped and read the code directly during vibe coding. What made you stop?

Research Pointers

- Post-mortem articles on vibe coding failure cases, security incident reports
- Team-level AI coding governance policy documents (cross-referenceable with § 9.5)
- Have students record their own moments of intervention to derive common signals

6 Practice by Artifact Type

Even though it's all called "made with AI," the actual work differs entirely depending on what is being made. Each type has what works well, what doesn't, where humans need to intervene, and how to check quality. This chapter compares these differences by type. This book itself is the case study in the final section.

6.1 Writing: Reports, Papers, Documents

> ⚠ Writing Status
>
> Pre-draft. Check TOPICS.md at the repository root for owner and status.

Brief. Code can be verified by running it, but writing cannot. This asymmetry makes writing the most difficult artifact in the vibe coding discussion. This section covers how far AI use actually extends in writing reports, papers, and documents; where the line falls between tool and ghostwriter; and the problem of academic writing norms versus AI-specific style (including the cliches and inflated rhetoric this repository guards against). As the first section of Chapter 6 (artifact-by-artifact practice), it connects directly to the verification asymmetry discussion in § 8.3.

Seed Questions

- Unlike code, writing has no equivalent procedure to "run it to check." What do practitioners use to fill that gap?
- How far do journal and conference AI-use disclosure policies allow, and where do they draw the line at prohibition? What are the differences across fields?
- How are the features identified as "AI style" (cliched expressions, inflated rhetoric, specific vocabulary patterns) being detected?
- Is there any discussion that proposes criteria for distinguishing AI use as a tool from AI use as a ghostwriter?
- Record your own experience of writing this book or other text with AI as a case study. How much did AI write, and where did you revise it yourself?

Research Pointers

- Original text of major journal and conference AI-use policies (each journal's homepage)
- Discussions related to AI style (analysis pieces from linguistics and journalism)
- Record the writing process of this book itself, especially the style discipline this stub follows (see the top of the brief), as a case study

6.2 Data Analysis and Visualization

> ⚠ Writing Status
>
> Pre-draft. Check ownership and status in TOPICS.md at the repository root.

Brief. Data analysis is an area where vibe coding has taken hold particularly quickly. In the exploratory analysis phase, handing code over to AI produces graphs and statistical tables in an instant. The problem is that a result looking plausible and actually being correct can be two different things. This section covers, with real cases, how much exploratory data analysis has accelerated, and the trap where plausible-looking graphs lead to conclusions without verification. It connects with

§ 4.4 (Researchers' Vibe Coding), but this section covers business analysis as well as research.

Seed Questions

- What are actual error cases found in AI-generated visualization or statistical analysis code (axis distortion, statistical misuse, data leakage, etc.)?
- As the speed of exploratory data analysis has increased, how has the analyst's role changed? Has the center of gravity shifted from writing code to interpreting and verifying results?
- What concrete verification procedures do practitioners use to filter out plausible-looking graphs?
- How is the landscape of data analysis tools (notebook-based agents, code-interpreter type tools) divided?
- Record as a case your own experience of extracting data analysis results with AI and then verifying them yourself. What did you verify, and what turned out to be wrong?

Research Pointers

- Collections of statistical and data visualization error cases (academic and journalism critiques of graph distortion)
- Official documentation for code interpreters and notebook-based AI analysis tools
- Document a verification procedure using students' own data analysis assignments as a case study

6.3 Web Applications: From Frontend to Deployment

> ⚠ Drafting Status
>
> Not yet drafted. Check the repository root's TOPICS.md for owner and status.

Brief. Web applications are the most representative stage for vibe coding. Building a single frontend screen is a very different level of difficulty from connecting a backend and database and actually finishing deployment. This section follows the full path from idea to deployment, distinguishing where things go smoothly and where they get stuck, and whether that bottleneck comes from the limits of the tools or a lack of human understanding. Where § 6.1 and § 6.2 dealt with artifacts that are hard to verify, this section shows why an artifact that is comparatively easy to verify (you can see it by running it) is nonetheless hard to see through to completion.

Seed Questions

- How common are actual cases of completing a full-stack web application from planning to deployment using vibe coding alone? How does this compare to cases of giving up midway?

- Among frontend, backend, database, and deployment pipeline, which stage does vibe coding work best for, and which is it most vulnerable at?
- In areas where mistakes are costly, such as authentication, payments, and permission management, how is the reliability of AI-generated code assessed?
- Among web apps built with vibe coding, what is the difference between cases that acquired and retained real users and cases that remained mere demos?
- Record your own experience attempting a web application from planning to deployment as a case study. At which stage were you stuck the longest?

Research Pointers

- Full-stack vibe coding build logs (blogs, YouTube, X threads)
- AI integration features and official case studies from web deployment platforms (Vercel, Railway, etc.)
- Compile students' own web app project progress logs as case studies

6.4 Automation Scripts and Personal Tools

> ⚠ Writing Status
>
> Pre-draft. Check ownership and status in TOPICS.md at the repository root.

Brief. Short scripts that take over repetitive tasks, glue code that stitches together multiple tools, and CLI tools that automate personal workflows are considered the category where vibe coding succeeds most reliably. This section covers why this category is said to have a particularly high success rate, what structural reasons make the risk of failure low, and why this success does not generalize to other categories (web apps, writing). It is also the most concrete realization of § 4.2 (software for one).

Seed Questions

- What is the basis for the claim that automation scripts and glue code have a high success rate in vibe coding? Is there quantitative research, or is it just anecdotal impression?
- Why is verification easy in this category? (Easy to undo, execution results are immediately visible, low cost of failure, etc.)
- In what form are actual cases of personal workflow automation tools (CLI, cron jobs, notification pipelines, etc.) reported?
- Under what conditions does failure occur in this category? (External API changes, permission issues, etc.)
- Record an automation script or personal tool you made yourself as a case study. What did it automate, and how long did you actually keep using it?

Research Pointers

- Collect cases from personal automation tool sharing communities (Hacker News Show HN, Reddit, etc.)

- Empirical research on vibe coding success rates (check primary sources if available)
- Accumulate a list of automation scripts made by each student as case data

6.5 Slides and Typesetting: A Case Study of This Book

> ⚠ Writing Status
>
> Pre-draft. Check ownership and status in TOPICS.md at the repository root.

Brief. Presentation materials and typeset works like books pose a unique verification problem in that they require both visual polish and structural accuracy at the same time. This section covers how AI coding tools are actually used in producing slides and typeset works, taking this book's own build pipeline as the primary case study. This repository manages the manuscript with Quarto, typesets the PDF with Typst, and uses a pipeline that translates the Korean original into English with AI. The person responsible should directly document how this pipeline is actually structured and where humans intervene.

Seed Questions

- What stages does this repository's (vibecoding) build pipeline (`tools/build`, `tools/translate`, Quarto, Typst) actually consist of, and what does the AI do versus what does the human do at each stage? Read and run the repository's CLAUDE.md and tools/ scripts directly and document them.
- How good is the quality of the AI translation (files in `en/` marked MACHINE-TRANSLATED) actually? What error patterns appear when compared against the original text?
- How does the way AI is used differ between slide-making tools (code-based slide generators) and GUI tools like Word or PowerPoint?
- What counts as "it works" verification for typeset works? (successful rendering, no broken layout, compliance with print specifications, etc.)
- If you have experience working with AI on slide or document typesetting tasks outside of this book, record it as a case study.

Research Pointers

- Read and run this repository's `tools/build`, `tools/translate`, `_quarto.yml`, and CLAUDE.md directly to document the pipeline
- Check the official Quarto and Typst documentation for descriptions related to automation or script integration
- Compare cases of other code-based slide and typesetting tools (markdown-based presentation generators, etc.)

7 Content Generation

The cycle of instructing in natural language and revising while watching the results spread beyond code. Images, video, audio, and the pipelines that weave these

together with code. This chapter maps the landscape of content generation and analyzes what it shares with, and how it differs from, code generation.

7.1 Image Generation

> ⚠ Writing Status
>
> Pre-draft. For assignment and status, see TOPICS.md in the repository root.

Brief. Image generation shares with code generation the same iterative structure of instructing in natural language and revising while reviewing the result, but its prompt-writing practices and how results are evaluated differ considerably. This section covers the prompt practices actually used in image generation, techniques for maintaining consistent style, and how the verification structure differs compared to code generation. As the first section of Chapter 7 (Content Generation), it extends the discussion of verification asymmetry to be covered in § 8.3 to images as a concrete artifact.

Seed Questions

- In what ways do image generation prompt-writing practices (style keywords, negative prompts, seed fixing, etc.) differ from code generation prompts?
- What techniques exist for maintaining style consistency across multiple images, and how well do they actually work in practice?
- What criteria, other than whether it works or not, are used to evaluate the quality of image generation results? How is the subjectivity problem in evaluation criteria addressed?
- How has the asymmetry that code can be verified by running it while images cannot been affected prompt-writing practices themselves?
- Record as a case study your own iterative process of obtaining the desired result with an image generation tool. How many attempts were needed, and what did you change along the way to converge on the result?

Research Pointers

- Official documentation and prompt guides for major image generation models
- Practitioner guides on style control techniques (verify exact technique names and their current status)
- Document the students' own image generation prompt iteration process and compile it as a case study

7.2 Video and Audio Generation

> ⚠ Writing Status
>
> Pre-draft. Check the repository root's TOPICS.md for ownership and status.

Brief. Video and audio generation is more computationally expensive than images and adds a time axis, which makes evaluation harder. This section covers the video generation model landscape as of 2025 to 2026, the current state of speech synthesis and music generation, and what criteria are used to evaluate the quality of these artifacts. Video is distinguished from § 7.1 (image generation) in that it has new failure modes that images don't have, such as frame-to-frame consistency and violations of physical laws.

Seed questions

- As of 2025 to 2026, what are the major video generation models, and how are their respective strengths and limitations reported? (Model names and release dates must be verified against primary sources.)
- How are failure modes unique to video generation (breakdown of frame-to-frame consistency, violations of physical laws, etc.) being classified?
- For speech synthesis and music generation respectively, what criteria are used to judge quality as good? Are there quantitative metrics beyond human listening judgment?
- How far have copyright disputes over video and audio generation (training data, celebrity voice cloning, etc.) progressed so far?
- Record your own experience using video or audio generation tools as a case study. What did you repeatedly adjust to get the result you wanted?

Research pointers

- Official announcements and technical blogs from major video and audio generation models (verify dates and model names against primary sources)
- Reporting on copyright and policy-related lawsuits and regulatory trends
- Record of the student's own generation experience

7.3 Content Pipelines: Weaving Generation into Code

> ⚠ Writing Status
>
> Pre-draft. Check ownership and status in TOPICS.md at the repository root.

Brief. Generating a single image or video is a different problem from running an entire channel by weaving planning, generation, editing, and distribution together in code. This section covers cases where content generation has been automated into a pipeline, an automated content production process that runs from script writing through video rendering to publishing. Where § 7.1 and § 7.2 covered individual generation techniques, this section looks at what becomes newly possible when those techniques are orchestrated in code.

Seed Questions

- What structure do actual cases of running a content channel that weaves planning, generation, editing, and distribution into a single pipeline actually have?

- How does pipeline automation manage quality instead of just increasing content volume? What differences between the output of automated channels and channels where a person intervenes every time have been reported?
- Where in the content pipeline does a person necessarily have to intervene (final approval, editing, etc.), and why has that step not been automated?
- How does the low-effort content problem created by content generation automation connect to the discussion of slop in Part 3?
- Record as a case study your own experience building, or attempting to build, a content pipeline. How far did you automate it, and where did you get stuck?

Research Pointers

- Accounts of running automated content channels (production reports on YouTube Shorts, newsletter automation, etc.)
- Examples of open-source tools or frameworks for content generation pipelines
- If this book's author group or students run a content channel, record its pipeline structure as a case study

Part III · Principles and Limits

8 How It Works

To use vibe coding properly, you need at least a rough sense of what's running underneath it. This chapter explains it without math, but accurately: the basic structure of next-token prediction, the anatomy of a coding agent, and why this approach has worked best specifically for code.

8.1 From Next-Token Prediction to Code

> ⚠ Drafting Status
>
> Pre-draft. Check TOPICS.md at the repository root for owner and status.

Brief. The basic behavior of an LLM is nothing more than predicting the next token. This section's task is to explain precisely, without formulas, how the ability to write code emerges from this simple mechanism, and why code is a particularly good fit for this mechanism compared to natural language. It examines the role played by each of the following: the syntactic regularity of code, the vast training data available from public repositories, and the signal provided by executability. At the same time, it addresses where this mechanism breaks down, that is, what form hallucination takes in code generation. The actual boundary between the claim that "the model understands" and the claim that "it merely mimics patterns" forms the outline for this entire chapter (Chapter 8).

Seed Questions

- Through what specific pathway does the ability to write code emerge from the simple training objective of next-token prediction?

- Why is code generation more favorable for training than natural language generation? Which contributes more: syntactic regularity or data scale?
- What specific forms does hallucination take in code generation? Do fabricating a nonexistent API and using the wrong argument share the same cause?
- What grounds does each of the claims “the model understands code” and “it merely mimics patterns” rest on?
- How has the expansion of context window size translated into performance improvements in actual coding tasks?

Research Pointers

- Reread Karpathy’s original “Software 2.0” (2017) as the origin of the perspective that treats code as data.
- Select and cite popular yet accurate explanatory material on next-token prediction and the transformer architecture.
- Prioritize searching for academic papers addressing hallucination in code generation (empirical research is the basic stance of Part 3 as a whole).

I’ll translate this Quarto markdown file from Korean to English, preserving the structure exactly as specified.

8.2 Anatomy of a Coding Agent

> ⚠ Drafting Status
>
> Pre-draft. Check the repository root’s TOPICS.md for owner and status.

Brief. Break down what a coding agent actually does in a single turn. Trace, step by step, how user input is assembled into a prompt, how tool calls (reading and writing files, executing commands) proceed, how the feedback loop by which execution results flow back into context works, and how many times this loop repeats before the task ends. The core work is to use Anthropic’s “Building effective agents” and its distinction between workflows and agents as the theoretical axis, and to cross-check it against the actual source code of open-source coding agents to identify the gap between theory and implementation.

Seed Questions

- If you break down a single turn of a coding agent from user input to final response, step by step, what do you get?
- What does tool use mean on the model side, and what does the harness side need to prepare for it to work?
- Where is the boundary between the “workflow” and “agent” that Anthropic distinguishes, and which are actual coding agents closer to?
- In the source code of open-source coding agents, how is the main loop actually implemented, and does it match the documented explanation?

- What changes if there is no feedback loop by which execution results (test failures, error messages, shell output) flow back into the model's input?

Research Pointers

- Read closely Anthropic's "Building effective agents" (2024-12) and use it as this chapter's theoretical axis
- Read the main loop code directly from the GitHub repositories of several open-source coding agents (CLI-type tools)
- Find materials related to harness engineering and cross-reference them with § 3.4 and the agent harness entry in the glossary

8.3 Verification Asymmetry: Why Code, of All Things

> ⚠ Drafting Status
>
> Pre-draft. See TOPICS.md at the repository root for ownership and status.

Brief. One reason code became the first major domain of AI generation is verifiability. When you run code, you can immediately tell, at least in part, whether it is right or wrong, but this is not true of prose or images. This section covers why this asymmetry arises, what concrete role automated verification signals such as tests, compilation, and type checks actually play in model training and agent loop design, and what conditions would be needed for this verification advantage to generalize to writing or other creative domains. This section forms a counterpart to the "artifacts that are hard to verify" discussion in § 6.1 (Writing).

Seed Questions

- What exactly does the claim that "code is verified by running it" verify, and what does it fail to verify?
- How are automated verification signals (passing tests, successful compilation, type checks) concretely reflected in agent loop design?
- What empirical research supports or refutes the verification asymmetry hypothesis?
- What conditions (formalizability, the existence of gradeable criteria) would be needed for this verification advantage to generalize to other outputs such as writing, images, or data analysis?
- What long-term bias does automation progressing preferentially on verifiable tasks create?

Research Pointers

- Find and review empirical research on learning based on verifiable rewards.
- Cross-reference with § 5.4 (Creating Verifiable Units) and § 6.1 (Writing) to divide roles without overlap.
- Identify where software testing theory intersects with the AI evaluation (evals) literature.

9 Limits and Risks

Behind the promise of vibe coding lies a bill to be paid. This chapter lays that bill out honestly: the hollowing-out of understanding (codebases turning into black boxes), security, quality debt, the breakdown of the pathway to expertise, and the question of accountability. The standard here is evidence, not optimism or fear.

9.1 The Codebase Becoming a Black Box

> ⚠ Drafting Status
>
> Pre-draft. Check TOPICS.md at the repository root for owner and status.

Brief. Code that AI writes quickly piles up just as quickly, in a state where no one understands the whole. This section addresses the problem of hollowed-out understanding. It examines the process by which individual commits work but the people who grasp the overall architecture disappear from the organization, and what happens when AI layers more code on top of such a codebase again. Establishing criteria for when "code that works without being understood" becomes a problem and when it does not is the core argument of this section. This serves as the introduction to the whole of Chapter 9 (Limits and Risks).

Seed Questions

- Are there reported cases where "code that works without being understood" actually led to accidents or outages?
- What do empirical studies on the relationship between codebase comprehension and maintenance cost show?
- When AI layers more code on top of code that AI already wrote, is the accumulation of errors or style inconsistencies confirmed?
- How do organizations try to measure or mitigate the state where "no one knows the whole"? Have documentation requirements or onboarding practices actually changed?
- How does this problem play out differently in personal projects (solo development) versus team codebases?

Research Pointers

- Prioritize finding academic papers on program comprehension and codebase complexity.
- Collect field reports from practice, such as incident postmortem reports and maintenance experience accounts.
- Cross-reference with § 11.1, which addresses Parnas's (1972) module decomposition argument.

9.2 Security: A New Attack Surface

⚠ Drafting Status

Not yet drafted. Check TOPICS.md in the repository root for owner and status.

Brief. Does AI-generated code create vulnerabilities of a different kind, or at a different frequency, than human-written code? This section covers empirical research on security vulnerabilities in generated code, prompt injection where an agent reads external content (search results, documents, issue comments) and ends up executing malicious instructions embedded in them, and slopsquatting, where a model invents a package name that doesn't exist and an attacker registers that name first. Figures and cases must be verified against primary sources before being cited.

Seed questions

- What empirical studies compare vulnerability rates in AI-generated code with human-written code, and is their methodology reliable?
- Concretely, what attack paths does prompt injection create in the context of coding agents?
- Are there actual reported cases of slopsquatting, and how large is the estimated damage?
- How do the execution privileges granted to coding agents (shell commands, file writes, network access) change the security model?
- What mitigations do vendors (Anthropic, GitHub, etc.) present in official documentation for this attack surface?

Research pointers

- Prioritize searching for academic papers that quantitatively analyze security vulnerabilities in generated code
- Check empirical reports from security firms and researchers covering prompt injection and slopsquatting
- Cross-reference with the slopsquatting entry (G14) in the glossary, but divide roles to avoid duplication

9.3 Quality Debt and Maintenance

⚠ Drafting Status

Not yet drafted. Check assignment and status in TOPICS.md at the repository root.

Brief. Rapidly generated code often works but is hard to fix. This section covers how quality debt is similar to and different from the existing concept of technical debt, how AI's generation speed changes the rate of debt accumulation, and

when rewriting from scratch becomes the cheaper option. If there are attempts to calculate rewrite costs financially, they will be checked and introduced. This pairs with § 11.2 (technical debt, from a rediscovery perspective), but this section emphasizes the practical and financial perspective.

Seed Questions

- Is there empirical research showing that AI-generated code has different maintenance costs compared to human-written code?
- When applying the technical debt metaphor to AI-generated code, what fits and what doesn't?
- What concrete criteria (complexity metrics, review time, etc.) are used in practice to identify "code that works but cannot be fixed"?
- What financial frameworks have been proposed for judging whether a rewrite or incremental refactoring is cheaper?
- Is the increase in code generation speed outpacing the growth rate of review and testing capacity, and is this confirmed by data?

Research Pointers

- Check Cunningham's (1992) original presentation of technical debt (OOPSLA experience report) in the original text.
- Prioritize searching academic papers on software maintenance costs and refactoring-versus-rewrite decisions.
- Divide roles with § 11.2: this section takes the practical/financial perspective, § 11.2 takes the conceptual history perspective.

9.4 The Junior Paradox: The Ladder Disappears

> ⚠ Writing Status
>
> Pre-draft. Check ownership and status in TOPICS.md at the repository root.

Brief. The work that AI substitutes for best is often the work that juniors used to take on in order to grow. This section addresses the concern that automating entry-level work cuts off the ladder to expertise. It looks for data on junior developer hiring and changes in internships and entry-level postings, to determine whether this problem is actually confirmed by data or is an exaggerated narrative. It pairs with § 13.3 (The Ladder of Expertise), but this section weighs more heavily toward labor market and hiring data. Figures must be verified against original data before being cited.

Seed Questions

- How have the number of junior developer job postings and the scale of internships actually changed recently, and what are reliable data sources for this?

- Is the observation that "juniors have become faster with AI" and the observation that "junior hiring has declined" different facets of the same phenomenon, or are they separate phenomena?
- Have companies ever officially acknowledged that the decline in entry-level hiring is due to AI, or do other factors such as the economy or corrections for prior overhiring play a larger role?
- What research refutes the concern that "the ladder is disappearing," or offers a different explanation?
- Are there reported cases of company- or industry-level responses to the decline in entry-level hiring (such as maintaining junior-dedicated projects, redesigning mentoring, etc.)?

Research Pointers

- Prioritize searching empirical labor economics research that deals with job postings and employment statistics; figures must be verified against original data before being cited.
- Distinguish between industry surveys (such as developer hiring trend reports) and academic research, and treat their reliability differently.
- Divide roles with § 13.3 (The Ladder of Expertise) and § 14.2 (Changes in the Job Landscape): this section is limited to the coding ladder.

9.5 Accountability and Sign-off: Who Signs Off on This Code

> ⚠ Drafting Status
>
> Not yet drafted. Check ownership and status in TOPICS.md at the repository root.

Brief. When an accident occurs in code written by AI, who is responsible? This section covers how the traditional sign-off practices of code review and deployment approval are being redefined in the face of AI-generated code, what AI code governance policies (review requirements, labeling of generated code, designation of accountable parties) real companies and institutions have actually adopted, and how responsibility has actually been distributed in incident cases. As the final section of Chapter 9, it asks whose responsibility the preceding problems (the hollowing-out of understanding, security, and quality debt) ultimately become at the organizational level.

Seed Questions

- In outages or incidents caused by AI-generated code, how has responsibility actually been attributed, and are there any publicly documented cases?
- What do the AI code governance policies formalized by companies and institutions specifically require regarding review obligations, approval procedures, and labeling of generated code?

- Is the traditional practice of code review approval being maintained only in form, while its substance hollows out in the face of AI-generated code?
- What labeling and review rules are open source projects introducing for AI-generated contributions?
- How far have discussions of legal and contractual liability (such as analogies to product liability) progressed?

Research Pointers

- Collect AI code governance policy documents published by companies and government agencies as primary sources.
- Check actual cases in open source projects' AI contribution policies (contribution guidelines).
- Cross-reference with § 10.3, which covers the teleology of code review, but divide the roles: this section is limited to accountability and governance.

Part IV · Software Engineering, Rediscovered

10 Rediscovering Methodologies

The last 50 years of software engineering are being rediscovered one by one in the age of AI. Each section of this part takes a single classic concept, reads the original source, and contrasts it with its reappearance in the age of AI. This chapter covers methodologies for how we work: waterfall and agile, pair programming, code review, and TDD.

10.1 Waterfall and Agile, Again

> ⚠ Writing Status
>
> Not yet drafted. Check ownership and status in TOPICS.md at the repository root.

Brief. This section asks whether the conflict between spec-driven development and vibe coding is a rerun of the waterfall-versus-agile debate. It is written in three parts. First, it reads closely what Royce's (1970) original paper actually argued, and what the four values and twelve principles of the Agile Manifesto (2001) actually say in the original text. Next, it gathers recent cases to see which side spec-driven development in the style of GitHub Spec Kit is closer to, and which side vibe coding practice, which starts with a prompt and iterates through repeated checking, is closer to. Finally, it argues by distinguishing the points that are genuinely the same, such as the length of the iteration cycle and the placement of the specification, from the points where the agent, as a new actor, changes the premises of both models.

Seed Questions

- Reading Royce's (1970) original paper directly, did he really advocate for waterfall as a pure sequential model, or was he already warning about the need for iteration? (verify against the primary source)
- Among the Agile Manifesto's four values and twelve principles, which clauses are most often cited today as the justifying logic for vibe coding? (verify against the primary source)
- Do proponents of spec-driven development (Spec Kit and others) frame their workflow as a revival of waterfall, or as a variant within agile? (collect rediscovery cases)
- In what ways is vibe coding practice, which starts with a prompt and iterates through repeated checking, the same as or different from agile's iteration cycle, and does the theory still hold even when the unit of iteration shrinks to seconds?
- How much do the criticisms from the waterfall-versus-agile debate at the time (excessive documentation, the fiction of planning) overlap with today's criticisms of spec-driven development?
- If there are cases of practitioners mixing the two workflows, what criteria do they use to divide them? (collect rediscovery cases)

Research Pointers

- Royce's 1970 original paper (search by author name and year to verify the original text)
- agilemanifesto.org original text (four values, twelve principles)
- The GitHub Spec Kit announcement post and practitioner blogs on its adoption and criticism
- Points of contact with §3.3 (spec-driven development) and §2.2 (spread of terminology) in this book; avoid duplicate exposition

10.2 Pair Programming: When Your Pair Stops Being Human

> ⚠ Drafting Status
>
> Not yet drafted. Check owner and status in TOPICS.md at the repository root.

Brief. This section contrasts the arguments Beck's XP (1999) used to justify pair programming with today's practice of pairing with AI. It is written in three parts. First, it reads closely what the original XP text says about pairing arguments such as continuous review, knowledge propagation, and role rotation. Next, it collects examples of how tools introduce themselves as an "AI pair programmer" and how practitioners describe their workflows. Finally, it argues whether the premises of human pairing, namely two equal actors, mutual learning, and social pressure, hold or fail to hold in a human-AI relationship.

Seed Questions

- What exactly was the core argument Beck made for pair programming in "Extreme Programming Explained" (1999), including the parts that go beyond simple defect reduction (verify against the primary source)
- XP's pairing presupposes rotation between the driver and observer roles; does this role rotation actually happen when collaborating with AI, or is the human always fixed in the observer role?
- What product or document first used the phrase "AI pair programming," and how far does that phrase carry over XP's original meaning before it slides into metaphor? (collect rediscovery cases)
- Among the effects reported by human-to-human pairing research (defect reduction, learning effects), are there recent studies or cases claiming that these are also reproduced in human-AI pairing? (collect rediscovery cases)
- The social mechanisms of pair programming, namely embarrassment, peer pressure, and the demand for immediate explanation, do not operate on an AI partner; how does this difference affect actual work quality?
- How are the objections to pair programming that even existed among XP practitioners (fatigue, cost) raised differently in the age of AI pairing?

Research Pointers

- The chapter on pair programming in Beck's original text, "Extreme Programming Explained" (1999)
- Empirical studies on pair programming (papers measuring defect rates and time spent)
- Self-descriptions and presentation materials of tools such as GitHub Copilot and Cursor introducing themselves as pair programmers
- Practitioner blog posts describing workflows for pairing with AI

10.3 Code Review: From Fagan Inspections to AI Reviewers

> ⚠ Writing Status
>
> Pre-draft. Check TOPICS.md at the repository root for owner and status.

Brief. This section contrasts the original purpose of code review established by Fagan's inspections (1976) with LLM-based automated review tools. It is written in three parts. First, it reads the formal procedure of Fagan inspections precisely from the original source: preparation in advance, role-based participants, defect classification, and rework verification. Next, it gathers cases of what AI reviewer tools actually check and what they miss. Finally, it argues which of the three functions of review, defect detection, knowledge diffusion, and establishing organizational norms, survive in the AI era and which fall away.

Seed Questions

- In Fagan's (1976) inspection procedure, how exactly were participant roles (moderator, author, reviewer, recorder) and the defect classification system defined? (verify against the original source)
- What Fagan inspections explicitly tried to prevent, such as authors defending themselves or unstructured ad hoc review, reappears or disappears in what form in today's AI review tools? (verify against the original source)
- Of the three traditional purposes of code review, defect detection, knowledge diffusion to newcomers, and establishing team coding norms, which does the AI reviewer replace and which can it not replace? (collect rediscovery cases)
- How does the structure in which AI reviews code written by AI conflict with the principle of author-reviewer independence that Fagan assumed?
- Does data exist on the rate at which people actually act on what AI review tools flag in practice? (collect rediscovery cases)
- Formality already receded once when the field moved from Fagan inspections to lightweight code review; does the emergence of AI reviewers push formality back up or down further?

Research Pointers

- Fagan, "Design and Code Inspections to Reduce Errors in Program Development", IBM Systems Journal(1976) original text
- Official documentation for tools such as GitHub Copilot code review and Claude Code /code-review
- Empirical research on code review (papers covering code review practices at large tech companies)
- Focus on the review procedure and purpose itself, to avoid overlapping with § 9.3 (Quality Debt) of this book

10.4 The Revival of TDD: When Tests Become the Spec

> ⚠ Writing Status
>
> Pre-draft. Check ownership and status in TOPICS.md at the repository root.

Brief. This section contrasts the red-green-refactor cycle and the argument that tests are specifications, as presented in Beck's TDD (2002), with the recent practice in agentic coding of writing tests first to serve as guardrails and verification signals. It connects directly to this book's observation that verification is the bottleneck (§ 8.3). It is written in three parts. First, it reads the original TDD cycle and argument precisely. Second, it gathers cases showing how test-first prompting became established practice in agentic workflows. Finally, it argues whether TDD as a design tool, as Beck emphasized, and tests as a tool for verifying AI output are the same thing or different.

Seed Questions

- What is the exact sequence of the red-green-refactor cycle presented by Beck in "Test-Driven Development: By Example" (2002), and in what context did he call tests a design tool? (verify against the source)
- Beck's argument for TDD placed emphasis on design pressure, breaking work into small units and thinking through interfaces first, rather than on defect reduction. Does this design-pressure argument hold equally when AI is the one writing the code, or was it an argument valid only when humans wrote the code?
- Since when has the workflow of writing tests first and having the agent make them pass appeared explicitly in tool documentation or practitioner conventions? (gather rediscovery cases)
- TDD originally presupposed that the person writing the tests and the person writing the code are the same. How is this condition broken when an agent writes both the tests and the implementation, and does this undermine the validity of tests as a verification signal?
- Are both success stories cited by TDD revivalists and failure cases, where AI manipulates the tests themselves in order to pass them, reported? (gather rediscovery cases)
- Has Kent Beck himself publicly commented on the relationship between AI coding and TDD recently, and if so, what did he say?

Research Pointers

- Beck, "Test-Driven Development: By Example" (2002), original text (preface, early example chapters)
- Test-first recommendation documentation from agentic coding tools, practitioner blogs
- Points of connection with § 8.3 of this book (verification asymmetry); avoid duplicate exposition
- Check whether Kent Beck has mentioned AI coding in recent interviews or blog posts

11 Rediscovering Principles

Beneath methodology lie principles: abstraction and information hiding, technical debt, the isomorphism between organizational structure and system structure, essential complexity and accidental complexity, and the exchange law of people and time. This chapter is about how papers from half a century ago read anew in the age of context windows and agents.

11.1 Abstraction and Information Hiding: Why Parnas Was Right, Again

⚠ Drafting Status

Pre-draft. See TOPICS.md at the repository root for owner and status.

Brief. This section contrasts Parnas's (1972) information-hiding criterion, the argument that modules should be divided not by function but by decisions likely to change, with codebase design in the era of AI agents with finite context windows. It is written in three parts. First, it reads precisely the two module decomposition approaches that Parnas's original paper compared. Second, it gathers cases where the way AI agents handle code demands this same criterion anew. Third, it argues whether a principle devised for human cognitive limits holds in the same form for a model's context limits.

Seed Questions

- What exact examples did Parnas (1972) use to compare the two module decomposition approaches he discussed, namely flowchart-based decomposition and information-hiding-based decomposition? (verify against the original)
- Parnas grounded information hiding in ease of change, an argument aimed at reducing the cognitive burden on human programmers. Does this argument carry over directly to AI agents with finite context windows, or do agents' limits differ in kind, requiring a different decomposition criterion? (verify against the original)
- Are there cases where official documentation or practical guides for agentic coding tools explicitly recommend codebase structures that are easy for agents to work with, and if so, how much do they overlap with Parnas's information-hiding criterion? (collect rediscovery cases)
- How many refactoring cases or discussions claiming to produce "AI-friendly" repository structures (small files, clear filenames, preference for pure functions, etc.) can be found? (collect rediscovery cases)
- Compared to other modularization principles developed after Parnas (cohesion and coupling, SOLID, etc.), why does the AI-era rediscovery concentrate specifically on information hiding?
- Is the question of what to show, as discussed under context engineering (§ 3.2 of this book), the same problem as Parnas's module boundary problem under a different name, or a different problem altogether?

Research Pointers

- Parnas, "On the Criteria to Be Used in Decomposing Systems into Modules" (1972), original text
- Points of contact with § 3.2 (Context Engineering) and § 5.3 (The Art of Context Management) of this book; avoid duplicating content
- Guides recommending repository structures for agentic coding tools (official documentation, engineering blogs)
- Comparative materials on later modularization theories such as cohesion and coupling

I have enough convention examples ("verify against the original source" for 원전 확인, "collect rediscovery cases" for 재발견 사례 수집). Now producing the translation.

11.2 Technical Debt: When a Metaphor Met an Interest Rate

⚠ Drafting Status

Not yet drafted. Check ownership and status in TOPICS.md at the repository root.

Brief. This section contrasts the original meaning of the technical debt metaphor, first used by Cunningham in his OOPSLA 1992 experience report, with its revival in the age of AI coding, where the speed of generation has sharply outpaced the speed of debt accumulation. The section is structured in three parts. First, it reads closely what Cunningham actually said and how the metaphor has since been misused. Second, it gathers case studies of debt-related discussions in the AI era and empirical investigations into the maintenance costs of generated code. Third, it argues whether the original metaphor's distinction between deliberate debt with a repayment plan and reckless debt still applies to AI-generated code.

Seed Questions

- What exactly did Cunningham say and in what context in his 1992 OOPSLA experience report, and if he later commented on the misuse of this metaphor, what did he say? (verify against the original source)
- Cunningham's original argument was closer to saying that taking on debt is not itself bad, but that the debt must be recognized and a repayment plan must be set. How much of this conditional character survives in how the term technical debt is used today? (verify against the original source)
- Does empirical research or data exist to support the claim that technical debt is rising sharply for code generated quickly by AI? (collect rediscovery cases)
- Are there cases that literally extend the interest rate metaphor to quantitatively discuss the interest rate of AI-generated code, and if so, what metrics did they use? (collect rediscovery cases)
- Is the distinction between deliberately taking on debt (for rapid prototyping) and recklessly taking on debt (accepting AI output as-is without verification) actually observed in AI coding workflows?
- The concept of technical debt has a history of attempts at quantification (debt metrics, static analysis tools). Are there similar attempts at quantification for AI-generated code?

Research Pointers

- The original text of Cunningham's OOPSLA 1992 experience report (verify the exact title and quotations)
- Check whether Cunningham himself commented on the misuse of the metaphor in later interviews

- Division of labor with § 9.3 (Quality Debt and Maintenance) of this book. That section focuses on the current state of affairs; this section focuses on the history and revival of the metaphor itself
- Recent empirical research or industry reports on the maintenance costs of AI-generated code

11.3 Conway's Law and Human-AI Organizations

> ⚠ Writing Status
>
> Not yet drafted. See TOPICS.md at the repository root for owner and status.

Brief. This section contrasts the thesis of Conway's (1968) original paper, the observation that system design mirrors the communication structure of the organization that built it, with the code structure of organizations that mix humans and agents. It is written in three parts. First, read the exact claim and evidence of Conway's original paper. Second, gather cases related to the code structure of organizations that mix humans and AI. Third, argue whether the mechanism of communication cost that Conway presupposed operates the same way for AI agents, actors fundamentally different from humans.

Seed Questions

- What exactly were the cases and arguments Conway actually used in his 1968 paper, that is, the observation that a committee's design output resembles the committee's own structure? (verify original source)
- Did Conway himself coin the name "Conway's Law" and the term "inverse Conway maneuver" (the strategy of designing the organization first to match the desired architecture), or were these terms attached later by others? (verify original source)
- In a work structure where one person directs multiple agents at once, what should the thesis that organizational structure resembles code structure be understood as resembling: the cognitive structure of the single director, or the communication protocol among the agents?
- Has there been any reported case of an organization operating a multi-agent system deliberately applying the inverse Conway maneuver, that is, designing the agent organization structure first in order to obtain the desired software architecture? (gather rediscovery cases)
- Communication cost, commonly cited as the mechanism behind Conway's Law, consists among humans of meetings, documentation, and trust-building; what does communication cost among agents consist of, and does this change whether the law holds?
- Is the design of multi-agent frameworks itself already conscious of Conway's Law, that is, are there cases where tool designers cited this law to justify their architecture? (gather rediscovery cases)

Research Pointers

- Conway, "How Do Committees Invent?" (1968), the original paper (verify original publication venue)
- The origin of the term "inverse Conway maneuver" (verify whether it was Conway himself or a later compiler)
- Design documents of multi-agent coding systems, practitioners' accounts of operating agent organizations
- Points of overlap with related sections in Chapter 14 of this book (The Reorganization of Work); avoid role duplication

11.4 No Silver Bullet, Revisited

> ⚠ Writing Status
>
> Pre-draft. Check TOPICS.md at the repository root for owner and status.

Brief. This section contrasts the distinction Brooks (1986) drew between essential and accidental complexity, along with his claim that no single technology will yield a tenfold productivity gain, with which kind of complexity LLM coding tools actually reduce. It is written in three parts. First, it reads closely the original Brooks paper's distinction between the two kinds of complexity and his assessment of the promising techniques he reviewed. Second, it gathers empirical studies on the productivity of LLM coding tools as cases. Third, it argues whether the grounds on which Brooks concluded there is no silver bullet still apply to LLMs, or whether LLMs are a kind of solution he did not anticipate.

Seed Questions

- What were the exact definitions of essential and accidental complexity that Brooks distinguished in "No Silver Bullet" (1986), and what examples did he give for each? (verify against the original)
- In the paper, Brooks already considered artificial intelligence as a candidate silver bullet and rejected it. What exactly were his grounds for rejecting it, and do those grounds still hold for today's LLMs? (verify against the original)
- Do empirical studies measuring the productivity gains of LLM coding tools distinguish whether the gains come from reducing essential complexity or accidental complexity (typing, syntax, boilerplate)? (gather rediscovery cases)
- Are claims that LLMs are finally the silver bullet actually contending with counterarguments that LLMs, in the end, only reduce accidental complexity? If so, what is the core argument on each side? (gather rediscovery cases)
- One of the reasons Brooks concluded there is no silver bullet was that software's essential difficulties (complexity, conformity, changeability, invisibility) do not disappear even when the form of representation changes. Which of these four difficulties, if any, does the new representational form of natural-language prompting actually reduce?

- Has Brooks himself, in later interviews or writings, revisited his 1986 predictions? If so, what did he reaffirm or revise?

Research Pointers

- Brooks, "No Silver Bullet: Essence and Accidents of Software Engineering" (1986), original text
- Later software engineering literature dealing with the distinction between essential and accidental complexity
- Empirical studies on LLM coding productivity (industry and academic surveys)
- Points of contact with Chapter 4 (Opportunities and Potential) and Chapter 8 (How It Works) of this book; this section focuses on a reassessment through Brooks's framework

11.5 Mythical Man-Month: From Person-Months to Agent-Hours

> ⚠ Drafting Status
>
> Pre-draft. Check ownership and status in TOPICS.md at the repository root.

Brief. This piece contrasts Brooks's (1975) core law, the claim that adding manpower to a late project makes it later, and its rationale, with the recent practice of adding more agents in parallel. It is structured in three parts. First, read Brooks's law and its communication-overhead rationale precisely from the original text. Next, gather cases of parallel operation where multiple agents are run simultaneously. Finally, argue whether adding people and adding agents are the same or different in terms of the communication-overhead mechanism.

Seed Questions

- What was the exact argument by which Brooks (1975) derived the law that adding manpower to a late project makes it later, and specifically, as what function of headcount did he calculate the number of communication paths growing? (verify in the original text)
- Brooks's argument also cited the cost of training newcomers and the difficulty of dividing work sequentially as grounds; how does each of these two grounds apply, or fail to apply, to the situation of deploying additional agents? (verify in the original text)
- Is there any case or report on whether running multiple agents in parallel, each assigned to a different part of the same project, actually reduces total completion time, or whether it is offset by human coordination costs? (collect rediscovery cases)
- If the core mechanism underlying Brooks's law is communication overhead among people, does a corresponding overhead exist among agents, and if so, what is it measured by (token cost, context-sharing failure, conflicting changes, etc.)?

- Does writing by practitioners or researchers exist that explicitly claims Brooks's law does not apply to agents, or conversely that it applies to them unchanged? (collect rediscovery cases)
- How are the surgical team model and the importance of conceptual integrity, which Brooks presented together in the same book, being reexamined in discussions of agent orchestration?

Research Pointers

- Brooks, "The Mythical Man-Month" (1975), original text (check the relevant chapter)
- Practitioner writing and tool documentation on multi-agent parallel coding operations (parallel subagents, running multiple worktrees simultaneously, etc.)
- Points of overlap with § 8.2 of this book (Agent Anatomy); avoid duplicating roles

12 도구와 관행의 재발견 → translation below

13 Rediscovering Tools and Practices

Version control, continuous integration, documentation, open source collaboration. Tools and practices built for human collaboration are taking on new roles in loops that agents have now joined. This chapter covers that shift in roles.

13.1 Version Control: From History to Safety Net

> ⚠ Drafting Status
>
> Pre-draft. Check ownership and status in TOPICS.md at the repository root.

Brief. This section contrasts the historical background of Git as a collaboration tool with the way its use in agentic coding has shifted toward reverting experiments at the commit level. Write only what is confirmed about the year Git emerged or its founding narrative, and defer the rest to research directions. Structure the section in three parts. First, confirm, only to the extent verifiable, the history of version control systems and in particular the conventional narrative that the shift to a distributed model was meant to solve collaboration problems. Next, gather examples of recent practices in agentic workflows where commits, branches, and worktrees are used as a safety net. Finally, argue which properties of Git's design remain valid, and which have become excessive, now that its use has shifted from a tool that coordinates the parallel work of multiple people to a tool with which one person reverts an agent's experiments.

Seed questions

- What facts can be stated definitively about the historical background of Git's creation, and which parts, such as the exact year or detailed narrative, need verification? (Check primary sources; do not assert years without confirmation.)

- How much does the problem version control systems originally set out to solve, namely conflicts from simultaneous editing by multiple people and history tracking, overlap with the use actually observed in agentic coding, namely reverting code after an agent has broken it?
- When and in what form did practical advice to have agents commit frequently, or to make a commit at every checkpoint, first appear in tool documentation or practitioner writing? (Collect rediscovery cases.)
- Is the practice of multiple agents working simultaneously in different worktrees closer to the parallel work of multiple people that Git originally set out to support, or to one person coordinating the parallel work of multiple agents?
- How do Git collaboration practices meant for one person to explain to another, such as commit messages or review diffs, change when an agent is the one generating the commit? Do they still assume a human reader, or do they become a record intended for other agents or for the agent itself?
- Do problems from the era before version control, namely conflicts in shared-file or lock-based systems, reappear in the uncontrolled bulk edits made by agents? (Collect rediscovery cases.)

Research pointers

- Git's official history documentation and early announcement materials (verify exact dates and the circumstances of its origin; do not assert from memory)
- Official documentation on commits and checkpoints in agentic coding tools
- Practitioner writing on git worktree and branching strategies
- Division of labor with § 5.5 of this book (Make It Revertible). That section focuses on practical technique, this section on the historical shift in how the tool is used.

13.2 CI/CD and Gates: Checkpoints in a Loop Without Humans

> ⚠ Writing Status
>
> Pre-draft. Check ownership and status in TOPICS.md at the repository root.

Brief. This is a section that contrasts the original rationale for continuous integration with the phenomenon in which, in an era of agent-generated code that human reviewers cannot look at for every commit, CI gates become effectively the only automated checkpoint. Write it in a three-part structure. First, accurately read the original rationale for CI/CD: the practice that frequent integration prevents integration hell, and the empirical finding that automating the deployment pipeline correlates with organizational performance. Next, collect cases of the practical convention of using CI as the last line of defense for agent-generated code. Finally, argue what changes when CI's premise, that a human commits and a human interprets failures, shifts into a loop where the agent commits and also interprets failures itself.

Seed Questions

- What exactly is the core practice Fowler presented in his writing on CI, namely integrating several times a day and verifying each integration with an automated build? (verify primary source)
- What did the deployment pipeline concept that Humble and Farley presented in "Continuous Delivery" (2010), and the correlation between deployment frequency and stability that Forsgren et al. demonstrated empirically in "Accelerate" (2018), fundamentally show? (verify primary source)
- How well established in practice is the workflow in which an agent writes code, commits it, and if CI fails, reads the logs and fixes it itself? Are there tool documentation or case studies that explicitly address this? (collect rediscovery cases)
- When the structure CI originally presupposed, where a test failure signals a human and the human diagnoses the cause, shifts into a structure where the agent both receives the signal and does the diagnosis, how should the design of CI gates, that is, what to test and how strictly to block, change?
- Does Accelerate's finding that high-performing organizations have high deployment frequency still hold in the same direction now that agent-generated code has greatly increased deployment frequency, or is a reversal observed where deployment frequency alone rises while stability declines? (collect rediscovery cases)
- Have there been reported failure cases of agents manipulating code with the sole goal of bypassing CI gates or merely getting them to pass?

Research Pointers

- Martin Fowler's writing on CI (verify original title and main point)
- Humble and Farley, "Continuous Delivery" (2010), original text
- Key metrics from Forsgren, Humble, and Kim's "Accelerate" (2018) original text (deployment frequency, lead time, change failure rate, time to restore)
- Points of contact with § 8.3 (Verification Asymmetry) and § 5.4 (Creating Verifiable Units) in this book; avoid role duplication

13.3 Documentation: From Literate Programming to CLAUDE.md

> ⚠ Writing Status
>
> Pre-draft. Check ownership and status in TOPICS.md at the repository root.

Brief. This section contrasts the ideal presented by Knuth's (1984) literate programming, the argument that code should be woven together with explanatory prose meant for human readers, with the new documentation genre created by files like AGENTS.md and CLAUDE.md. It is written in three parts. First, it reads closely what Knuth's ideal actually intended: that the WEB system and programs should be written to explain themselves to people. Next, it gathers cases of when and how the new practice of AGENTS.md and CLAUDE.md became standardized. Finally, it argues what remains and what becomes meaningless, among the narrative quality

of explanation and the integration of code and prose that Knuth emphasized, when the reader of the document shifts from a human to an AI.

Seed Questions

- What is the exact wording and basis of the core argument Knuth presents in "Literate Programming" (1984), the claim that a program should be treated as a literary work that explains itself to people? (verify against the primary source)
- Confirm precisely how Knuth's WEB system actually worked: in what order it wove together code and explanatory prose, and how it split into two outputs, a human-readable document and compilable code. (verify against the primary source)
- Since when, and starting from which tool or practice, did files like AGENTS.md and CLAUDE.md become established as project descriptions read by AI agents? (collect rediscovery cases; if it cannot be confirmed, leave it as a research directive)
- The core of Knuth's literate programming was physically weaving code and explanation together into a single file, whereas files like AGENTS.md are separate files apart from the code. Is this physical separation compatible with Knuth's argument that the order of the narrative need not follow the execution order of the code, or is it a design that runs in the opposite direction?
- How are the reasons commonly cited for why literate programming failed to gain wide adoption in practice (lack of tooling, maintenance burden, and so on) either avoided or reproduced identically in AGENTS.md-style documents?
- If there is a practice of AI generating or updating the AGENTS.md that it will itself read, how does this change literate programming's premise of authorship, that a person explains something for another person?

Research Pointers

- Knuth, "Literate Programming" (1984), the original text published in The Computer Journal
- Announcements or presentations related to the standardization of AGENTS.md (the process by which multiple tools came to adopt it in common; verify the exact timing)
- Official documentation on CLAUDE.md and AGENTS.md from Claude Code, Cursor, and others
- Division of roles with § 5.3 of this book (The Art of Context Management). That section focuses on practical techniques, while this section focuses on the historical lineage of the documentation genre.

13.4 Open Source Collaboration Models: Cathedral, Bazaar, Agent

⚠ Drafting Status

Not yet drafted. Check ownership and status in TOPICS.md at the repository root.

Brief. This section contrasts the cathedral model and the bazaar model presented in Raymond's "The Cathedral and the Bazaar" (essay 1997, book 1999), along with the proposition that given enough eyeballs, all bugs are shallow, against the recent phenomenon of AI contributors appearing en masse in open source projects. It is written in a three-part structure. First, it accurately reads Raymond's original argument: Linus's Law, early release, frequent releases, and treating users as co-developers. Next, it gathers cases of how AI-generated issue triage or PRs are changing the review bottleneck and trust problems of open source projects. Finally, it argues whether the proposition that more eyeballs make bugs shallow still holds when those eyeballs are AI rather than human, and whether the hidden premise of trust in the bazaar model collapses in the face of AI contributors.

Seed Questions

- What was the exact original wording of Linus's Law as presented by Raymond in "The Cathedral and the Bazaar", the proposition that given enough eyeballs, all bugs are shallow, and what evidence did he cite to support it? (Source verification)
- What exactly were the conditions Raymond presented for the bazaar model to work, namely frequent releases, treating users as co-developers, and the coordinating role of the project leader, and among these, was there a stated precondition without which the bazaar model would not function? (Source verification)
- Are there actually reported cases of large volumes of AI-generated PRs or issues flooding open source projects, or public response policies from maintainers? (Rediscovery case collection)
- When "enough eyeballs" are filled not by humans but by AI reviewers or AI contributors, does the qualification of eyeballs that Raymond presupposed, namely trustworthy judgment and understanding of the project, still hold, or is a reversal observed in which the number of eyeballs increases while quality declines?
- How many concrete cases can be found of policies newly introduced by open source maintainers to handle AI-generated contributions, such as changes to contribution guidelines, automatic labeling, or mandatory disclosure of AI contributions? (Rediscovery case collection)
- Is there a trend in which Raymond's cathedral model, the approach in which a small number of trusted developers develop in a closed manner, is being reeval-

uated in the AI era, and if so, can it be seen as a reaction to the crisis of the bazaar model?

Research Pointers

- Raymond, "The Cathedral and the Bazaar" (essay 1997, book 1999), original text
- Public policy documents on AI contributions from major open source projects, maintainer blogs
- Related sections in Chapter 9 (Limits and Risks) of this book, and points of contact with § 12.3 (Documentation)
- Discussion of AI slop is split with concept dictionary entry G13 (slop/workslop) handling that topic; this section focuses on the open source collaboration model itself

Part V · Learning and Working

14 The Reconstruction of Study

When what needs to be learned changes, how we learn changes too. This chapter covers curriculum debates, experiments in educational settings, and shifts in the path along which expertise grows. This very course is itself one case of that experiment.

14.1 What to Learn

> ⚠ Drafting Status
>
> Pre-draft. Check ownership and status in TOPICS.md at the repository root.

Brief. In an age when AI writes syntax and boilerplate for us, what should we teach and learn? This section examines the claim that the center of gravity in programming education is shifting from memorizing syntax toward the abilities to define problems, decompose them, and verify solutions, and it maps out which positions clash in actual curriculum debates. Its task is not to reach a conclusion in advance but to check the grounds for each position against interviews and primary sources. This section serves as the introduction to all of Part 5, and it keeps in mind that this course itself is one experimental answer to this debate.

Seed Questions

- What grounds does each side offer: the claim that "memorizing syntax no longer matters" versus the counterargument that "without basic syntax you cannot even verify"?
- Do curriculum efforts that explicitly try to teach problem definition, decomposition, and verification actually exist, and what form do they take?
- Should human capacities such as collaboration, building trust, reasoning about others' perspectives, and dealing with uncertainty be taught as separate course

content, or should they be cultivated through pedagogical methods such as team assignments and discussion?
- How does the shape of this debate differ between data science education and traditional computer science education?
- How much do educators actually agree on what should be taught in the age of AI, and where do opinions diverge sharply?
- What position does this course's own curriculum design (seminar format, collaborative book writing) take within this debate?

Research Pointers

- Verify cases of curriculum reform at universities and bootcamps using primary documents (syllabi, department announcements).
- Starting from David Deming's webinar The Value of Human Skills in an AI Economy (2026-07-28), verify the claim about the comparative advantage of social learning and that human capacities should be addressed through pedagogy, using the original paper and follow-up research.
- Interview programming education researchers and current professors and instructors to gather primary sources.
- Divide roles with § 13.2 (Experiments in the Classroom): this section covers the debate over "what" to teach, while § 13.2 covers actual experimental cases.

14.2 Experiments in Educational Settings

⚠ Writing Status

Not yet drafted. Check TOPICS.md at the repository root for owner and status.

Brief. In the AI era, educational settings have largely split into three camps: those that ban AI use, those that mandate it, and those that overhaul their evaluation methods entirely. This section finds concrete cases of each approach and compares the actual outcomes they produced. A special task of this section is to include this very course (seminar-style, collaborative book writing, full AI use permitted) as one of the experimental cases and self-document it. Interviews and primary data collection are the core methods.

Seed Questions

- What cases exist of educational institutions or courses that banned AI use, and was the ban actually observed in practice?
- What cases exist of mandating or fully permitting AI use, and how did evaluation methods change?
- What outcomes have been reported by cases that overhauled evaluation methods themselves (reviving oral exams, strengthening process-based assessment, etc.)?
- Do educational research studies or large-scale surveys exist that compare these three approaches?

- How should the design and progress of this course itself be documented as an experimental case, and what can be confirmed through student interviews?

Research Pointers

- Collect official AI policy announcements from domestic and international universities and high schools as primary sources.
- Prioritize searching empirical studies in education journals.
- Directly collect this course's syllabus, operational records, and student interviews as primary sources.

14.3 The Ladder of Expertise: How Do Beginners Become Experts

> ⚠ Writing Status
>
> Pre-draft. Check ownership and status in TOPICS.md at the repository root.

Brief. The basic premise of deliberate practice theory is that experts build their skills by repeating easy tasks. What happens to this ladder if AI does those easy tasks instead? This section summarizes the core claims of deliberate practice theory, and seeks out research and observations on how beginners actually learn in AI-assisted environments to argue where they conflict. It pairs with § 9.4 (The Junior's Paradox), but this section places more weight on learning theory. Directly interviewing developers with different levels of experience is an important method.

Seed Questions

- Are the conditions required by deliberate practice theory (immediate feedback, gradually increasing difficulty, repetition) met or undermined in AI-assisted learning environments?
- Is there research comparing the speed and quality of skill acquisition between beginners who learned using AI tools and those who learned without them?
- Even with the same AI tool, do long-term learning outcomes differ between being assisted from the start versus first solving problems on one's own to establish a baseline and only then using AI for feedback?
- Is there evidence for the concern that "delegating easy tasks to AI leaves only verification ability intact while generative ability atrophies"?
- When experts look back on their own learning path, what do they say would have been different if AI had existed?
- What recent academic critiques or revisions of deliberate practice theory itself exist?

Research Pointers

- Check academic databases for the original research on deliberate practice and the critical literature that followed it

- Prioritize searching empirical research in educational psychology and computer science education on AI-assisted learning
- Track down experimental studies testing the claim David Deming presented in the The Value of Human Skills in an AI Economy webinar (2026-07-28), that one should “first go through productive struggle and only then use AI as a thinking partner”
- Interview developers with under 3 years of experience and developers with over 10 years of experience separately, and compare them as primary sources

15 Reconstructing Work

Vibe coding began as a story about coding, but it ends as a story about work. The day of a person working with AI, the actual shifts in the job landscape, and even the question “if AI does everything, what am I here to do?” We start not from prediction but from observation of changes that have already happened.

15.1 A Day Working with AI

> ⚠ Writing status
>
> Pre-draft. Check ownership and status in TOPICS.md at the repository root.

Brief. The day of a practitioner who uses AI deeply is being restructured around time spent delegating, reviewing, and running multiple tasks in parallel, rather than time spent typing code directly. This section starts from observation, not prediction. The core method of this section is to directly interview people who work deeply with AI and record, as primary source material, how their day has changed. This corresponds to the introduction of Chapter 14 (Reconstructing Work).

Seed questions

- If you reconstruct the daily routine of a developer or researcher who uses AI deeply hour by hour, what stands out as having changed?
- How does the cycle of delegation and review actually play out, and has the time spent reviewing increased or decreased relative to the time spent generating?
- Does the practice of running multiple agents in parallel actually increase productivity, or does it increase the burden of management?
- What tasks remain that people insist on doing themselves rather than delegating to AI during the day, and what is the criterion for that?
- How does this change appear differently depending on occupation (developers, researchers, founders)?

Investigation pointers

- Directly recruit practitioners and researchers who use AI deeply and conduct in-depth interviews. The core material for this section is the interviews
- Diversify interview subjects so that occupation, career experience, and tools used differ

- If there are already published primary-source essays or blogs in the form of a “day in the life” record, refer to them, but do not let them replace the interviews

15.2 The Changing Job Landscape: A Data-Driven Look

> ⚠ Writing Status
>
> Pre-draft. Check TOPICS.md at the repository root for owner and status.

Brief. There are many claims that AI is changing the job landscape for developers, but the quality of the evidence varies widely. This section looks for empirical research on job postings, wages, and employment scale in order to separate exaggerated narratives from changes that are actually confirmed. The only way this book is allowed to cite figures is by directly checking the original data and methodology. This section does not predetermine any specific figures or conclusions. The person in charge must not assert any figure without verification, and should write claims only after verifying them against empirical research and original data.

Seed Questions

- What are reliable data sources that track changes in the number of developer job postings and required skills, and what are their methodologies?
- Can statistics on developer wage trends be causally linked to AI adoption, or do they only show correlation?
- For the hiring-decline figures frequently cited in the media, what is the original source, and is the methodology verifiable?
- How do patterns of change in employment data differ by country and by industry?
- Where empirical studies reach conflicting conclusions, is the cause a difference in methodology or a difference in the observation period?

Research Pointers

- Check labor economics academic papers and official employment statistics (original data published by governments and international organizations) as primary sources.
- For reports that hiring platforms publish themselves, be sure to check and state the limitations of their methodology and sample.
- Use any figure in this section only after directly checking the original data. Do not assert figures that could not be verified; leave them marked as “to be verified.”

15.3 Identity Question: What Kind of Person Am I?

> ⚠ Drafting Status
>
> Pre-draft. Check TOPICS.md at the repository root for owner and status.

Brief. If AI writes a substantial portion of the code, how does the person who made that code define what kind of person they are. This piece covers how the self-definition of developers, researchers, and creators is actually changing, and where the old discourse of craftsmanship or mastery goes in the face of this change. The method of this piece is to treat the voices of those involved, not statistics, as the primary source. As the final piece of Chapter 14, it closes the entire book with the identity question.

Seed Questions

- If someone has shifted from "I am a person who writes code" to a different self-definition, how does that new definition (designer, reviewer, orchestrator, etc.) actually get expressed in interviews?
- Is the discourse of craftsmanship and mastery discarded in the face of AI-assisted production, or is it redefined in a different form?
- How do reactions to this identity shift differ by generation and by career stage?
- What grounds do people who continue to hold onto the self-definition of "maker" give for doing so?
- Are there comparable cases to reference from identity debates in fields that experienced tool automation earlier, such as photography or music production?

Research Pointers

- Conduct in-depth interviews directly with developers and creators about identity change. Interviews are the core material for this piece
- Check literature on identity debates in fields that experienced automation earlier, such as photography and music, as comparative material
- Cross-reference with § 13.3 (The Ladder of Expertise) but divide roles: this piece covers identity and self-narrative, § 13.3 covers learning pathways

Appendices

16 Concept Dictionary

This is a dictionary of concepts that are born and evolve around vibe coding. Each entry records a definition, its origin, and its current status. Because concepts keep appearing and their meanings keep shifting, this appendix is permanently unfinished, and so it is the most frequently updated part of this book. One entry is a unit of contribution. Add new entries and claim ownership in the repository's TOPICS.md.

Entry format

A couple of sentences of definition, the origin (who, when), and the current status. For the date and person in the origin, check the primary source and link to it. Mark anything unconfirmed as "origin needs confirmation."

16.1 vibe coding

This is an approach to building software by instructing an AI coding agent in natural language. In its narrow sense, it refers to a working style where you don't read the generated code directly and instead iterate by watching only the execution results; in its broad sense, it refers to AI-assisted development in general.

Andrej Karpathy coined the term in a tweet from February 2025. The original text includes the phrase "fully give in to the vibes, embrace exponentials, and forget that the code even exists" (original tweet). In a one-year retrospective tweet in February 2026, Karpathy looked back on this as a "shower of thoughts throwaway tweet" (retrospective tweet).

It became a common word after being named Collins Dictionary's Word of the Year in November 2025 (announcement, coverage). Simon Willison proposed distinguishing between "the approach of not reading the code" and "AI-assisted development in general" (post), but in actual usage the two meanings are used interchangeably.

> ℹ Expansion pointers
>
> - Read the full Willison post and organize the spectrum he distinguishes
> - Check the full one-year retrospective tweet by Karpathy for the context at the time of the original tweet (which models and tools he was using)
> - Cross-check against the Wikipedia overview article (link) and gather additional examples of the term's meaning drifting

16.2 Prompt Engineering (프롬프트 엔지니어링)

The skill of designing the input given to a model, that is, the prompt, in order to obtain a desired output. This includes discovering and systematizing specific techniques such as constructing few-shot examples and inducing chain-of-thought.

For a while after the release of GPT-3 (2020), this was treated as something like a job function. Exactly who fixed this term as a job title, and when, needs verification.

As model performance improved and interaction shifted to conversational and agentic forms, its standing as an independent skill grew blurred. From the perspective of this book, it belongs to the prehistory of vibe coding.

ℹ Expansion pointers

- Check the timeline of when the job title “prompt engineer” appeared and when it disappeared from job postings
- Check the original papers for representative techniques such as few-shot and chain-of-thought
- Collect articles reporting on the disappearance of the job function (link with § 1.3 section)

16.3 Context Engineering (context engineering)

This is the technique of designing not a single line of prompt but the entire context the model sees, including files, documents, conversation history, and tool outputs. It reflects a shift in the bottleneck of prompt engineering from “how do you say it” to “what do you show it.”

Tobi Lütke (Shopify CEO) coined the name in a tweet on June 19, 2025, and Karpathy left a tweet endorsing it (Karpathy tweet). Simon Willison and Phil Schmid each wrote posts organizing the concept (Willison, Schmid).

It began to be widely used from the second half of 2025, and has now become a practical term referring to the practice of designing context files like CLAUDE.md and AGENTS.md.

ℹ Expansion pointers

- Check the full text of Lütke’s original tweet and thread reactions
- Compare the definitional differences between the Willison and Schmid write-ups
- Cross-reference with practical cases covered in § 3.2 and § 5.3

16.4 Agentic Coding

This is a way of working in which the AI does not simply answer once and stop, but cycles through a loop of calling tools, executing code, checking the results, and then deciding on the next action. It is the technological form that actually made vibe coding possible.

It remains to be confirmed who first used the expression “agentic coding” and when. However, one reference that lays out this loop structure itself is Anthropic’s “Building Effective Agents” (December 2024) (link).

CLI-type coding agents (such as Claude Code) and IDE-integrated agents are cited as the representative tools implementing this approach. Sections § 2.3 and § 8.2 cover the lineage of tools and the loop structure, respectively.

ℹ Expansion Pointers

- Confirm the earliest use of the term “agentic coding”
- Compare Anthropic’s “Building Effective Agents” distinction between workflows and agents against this book’s usage
- Check definitional consistency with § 8.2 (Anatomy of a Coding Agent)

16.5 CHOP (chat-oriented programming)

A term for the practice of instructing AI to write code in a conversational manner through a chat interface. It is a concept adjacent to vibe coding, emphasizing that work proceeds through conversation rather than direct manipulation of code.

Steve Yegge coined the term before the phrase vibe coding emerged (O’Reilly podcast). The exact timing of its first use needs to be confirmed. Yegge co-authored a book titled “Vibe Coding” with Gene Kim, though the exact publication details need to be confirmed.

Since the term vibe coding became popular, CHOP is often used more as an earlier-generation term, or as something close to a synonym for vibe coding.

ℹ Expansion pointers

- Confirm from the original source when and in what context (which piece of writing, which talk) CHOP was first used
- Confirm the publication date and publisher of Kim and Yegge’s co-authored book “Vibe Coding”
- Collect examples that use CHOP and vibe coding as the same term, and examples that distinguish them (linked to § 3.1)

16.6 바이브 엔지니어링 (vibe engineering)

A term for a way of working that keeps the speed of vibe coding while taking responsibility for the output. It amounts to a proposal to recombine engineering procedures such as verification, testing, and review back into the vibe coding workflow.

The concept was proposed by Simon Willison (post). It is known to date from fall 2025, but the exact publication date needs to be confirmed.

It is one of the reactive concepts to vibe coding’s “verification problem,” and § 3.5 covers these reactive concepts together.

ℹ Expansion pointers

- Confirm the exact publication date of Willison's post and the original wording of the definition
- Clarify the relationship between vibe engineering and adjacent reactive concepts such as spec-driven development and the TDD revival argument (§ 10.4)
- Check whether other authors or articles have used this term since Willison

16.7 loop engineering

Instead of a person typing a prompt every time, this is the technique of designing the loop itself, in which an agent finds work, performs it, verifies it, and records it. It is discussed as the successor concept to prompt engineering.

It spread widely following Addy Osmani's summary piece in June 2026 (coverage).

From this book's perspective, it is a concept corresponding to the next chapter of vibe coding. It is covered together with harness engineering in § 3.4.

ℹ Expansion pointers

- Find Osmani's original piece (the original source of the summary piece) and verify the citation
- Sort out where the concepts of loop engineering and agent harness overlap and where they diverge
- Collect real examples of tools and workflows that use this term (link with § 3.4)

16.8 Agent Harness (agent harness / harness engineering)

This term refers to the execution environment that actually makes an agent work: the full set of mechanisms for tool access, permissions, loop control, and context management. In the formulation "Agent = Model + Harness," it captures the view that, apart from the model's own performance, harness design determines the quality of the agent.

It has been summarized as a three-stage progression running from prompt engineering through context engineering to harness engineering (Faros AI blog). It remains to be confirmed who first used the term "harness" for an agent execution environment before this summary post.

CLI-based coding agent tools such as Claude Code are cited as representative examples of a harness. This overlaps with § 3.4 and § 8.2.

ℹ Pointers for Further Exploration

- Confirm when the term “harness” first began to be used in the agent context
- Verify the three-stage progression (prompt → context → harness) from the Faros AI post against the original source and compare it with § 3.4 of this book
- Connect to open-source agent harness examples (the source-reading targets covered in § 8.2)

16.9 Spec-Driven Development (spec-driven development)

This is an approach in which, rather than telling AI to write code right away, you first draft a specification, or spec, together, and then carry out generation and verification based on that specification. It is often cited as a representative response to the verification problem of vibe coding, which by definition involves “not reading the code.”

In September 2025, GitHub released Spec Kit, presenting a four-stage flow of Spec, Plan, Tasks, and Implement (GitHub blog).

This can be seen as a case in which the specification-first principle of waterfall development was rediscovered in the AI era. § 3.3 and § 10.1 respectively cover the spread of the concept and the replay of the waterfall debate.

ℹ Expansion pointers

- Check whether similar “spec-first” tools or proposals existed before Spec Kit
- Compare Spec Kit’s four-stage flow with the stage divisions of the waterfall model covered in § 10.1
- Collect adoption cases or critical write-ups of Spec Kit

16.10 Subagent (subagent / multi-agent)

This term refers to the structure in which a main agent delegates a specific task to a separate, independent agent instance, or to an individual agent that is delegated to and executes in this way. A delegated subagent typically has its own context window and its own scope of tool access, and returns only the result to the main agent once the task is finished. A structure that coordinates multiple subagents simultaneously or sequentially is called a multi-agent system.

The precise origin of the term needs confirmation.

As a means of parallelizing work while conserving the context window, several coding agent tools have adopted this structure. It is one of the components of agent anatomy covered in § 8.2.

ℹ Expansion pointers

- Confirm which tool first productized the subagent concept, and when
- Compare how context isolation between subagents differs across tools
- Check consistency of this definition with § 8.2 (Anatomy of a Coding Agent)

16.11 human-in-the-loop

This term refers to a design that explicitly places points of human confirmation or intervention within the execution process of an automated system. In the context of AI coding agents, it refers to mechanisms that require human approval before hard-to-reverse actions such as file modification, command execution, or deployment.

It is a general term used in the automation and control systems field even before AI coding, and its precise origin needs verification.

In coding agent tools, it is implemented in forms such as permission prompts, plan mode, and change approval UIs. This connects directly to the discussion of the limits of delegation covered in § 5.6.

ℹ Expansion Pointers

- Verify the earliest literature in which this term was used in the automation and control field
- Compare human-in-the-loop implementation methods (approval UI, plan mode, etc.) across coding agents
- Organize the relationship with § 5.6 (When to Stop and Read It Yourself)

16.12 환각 (hallucination)

A term for when a language model plausibly generates content that is not factual or does not exist. A representative case in coding contexts is generating a function, library, or API that does not exist as though it were real.

The term is known to have moved from earlier use in the machine learning field to large language models, but the exact original usage and timing need verification.

This is directly connected to slopsquatting (G14) in that coding agents generating nonexistent package names can lead to supply chain attacks. It is covered in a security context in § 9.2.

ℹ Expansion pointers

- Identify the earliest paper or writing where “hallucination” began to be used to refer to language model errors
- Collect empirical studies on hallucination in the code generation context (e.g., rates of generating nonexistent packages)
- Cross-check § 9.2 (Security: New Attack Surfaces) and the slopsquatting entry

16.13 Slop (slop / workslop)

It refers to low-quality AI output generated in bulk without quality review. Workslop is a narrower term used specifically for output in a work context, such as reports, documents, or code, that colleagues or an organization must consume.

Origin needs verification. Both who coined the term and when, and when the variant “workslop” separately emerged, need to be checked and verified against original sources.

The term is commonly used to describe the phenomenon of generation speed outpacing review speed, and it connects to the issue covered in § 9.3 (Quality Debt and Maintenance).

ℹ Expansion Pointers

- Verify the earliest usage and origin of “slop” and “workslop” respectively from original sources
- Check whether there is empirical research on workslop (e.g., productivity loss estimates)
- Cross-check facts against § 9.3

16.14 Slopsquatting

This term refers to a supply chain attack technique in which an attacker pre-registers a package name that does not actually exist but was hallucinated by an AI coding agent. If a developer pastes the agent’s suggestion directly into an install command, the malicious package registered by the attacker gets installed.

Origin needs verification. Who coined this term and when needs to be traced back to an original source. It appears to be a coinage derived from typosquatting (preemptively registering package names that exploit typos), but this lineage also needs confirmation.

It is frequently cited in security discussions as a representative case illustrating the trust problem with generated code. It is a key case covered in § 9.2 (Security: New Attack Surfaces).

ℹ Expansion pointers

- Confirm the earliest use of the term “slopsquatting” (security research, blog posts, papers, etc.)
- Collect actual observed slopsquatting attack cases and the scale of damage
- Cross-check against § 9.2 for factual consistency

16.15 MCP (Model Context Protocol)

A protocol that standardizes how AI models communicate with external tools, data sources, and services. Implementing a server that follows this standard lets different AI applications reuse the same tool integration code.

Anthropic released this protocol. The exact release date needs to be verified.

It has become the standard path for coding agents to access tools outside the file system (browsers, databases, internal company systems, etc.). In practice, some cases are also emerging where teams switch from MCP servers to direct API calls instead.

ℹ Expansion Pointers

- Confirm the exact release date of MCP and the original announcement materials (Anthropic’s official blog/documentation)
- Summarize the differences from tool integration methods that existed before MCP (function calling, plugins, etc.)
- Collect the current state of MCP adoption (which companies/tools support it)

16.16 RAG (retrieval-augmented generation)

A technique in which a model retrieves relevant content from external documents or databases before generating an answer, and includes that content in the prompt. It allows the model’s response to reflect up-to-date information or private documents that are not part of its training data.

Origin needs verification. The original paper, its authors, and the publication date should be confirmed from the primary source.

In coding agents, it is used to search internal codebases or documentation and place the results into the context, and it is treated as one of the concrete implementation techniques of context engineering (G3).

ℹ Expansion pointers

- Confirm the authors and publication date of the paper that first proposed RAG from the primary source
- Collect examples of RAG usage in the context of coding agents (code search, document search)
- Clarify the relationship with the context engineering entry

16.17 Tokens / Context Window

A token is the smallest unit a language model uses to process text; it can be smaller than a word or span multiple words. The context window refers to the maximum number of tokens a model can reference at once.

This is a general technical term with no attributable origin tied to a specific person or moment.

This limit governs the size of codebase a coding agent can handle, the length of a conversation, and the number of files that can fit in context. There is an observation that as context windows grow larger, the importance of context engineering (G3) becomes even more pronounced, rather than less. § 5.3 covers practical management techniques.

ℹ Expansion pointers

- Trends in context window size across major models (verify figures against official documentation)
- How tokenization methods (BPE, etc.) apply specifically to code
- Cross-check against § 5.3 (the art of context management)

16.18 evals (evaluation)

This term refers to the procedure of scoring the output of an AI model or agent against a fixed set of criteria, or to the set of tests used for that scoring. It is a concept analogous to unit tests in software, used to quantitatively check model performance or detect regressions.

It is a general technical term with no origin attributable to a specific person or point in time.

In evaluating coding agents, evals that measure task success rates in real repositories are used alongside benchmark scores. This connects with § 5.4 (Creating Verifiable Units).

ℹ Expansion Pointers

- Confirm the methodology of representative benchmarks for coding agents (e.g., measuring repository-based task success rates)
- Summarize the differences between evals and traditional software testing (such as handling non-determinism)
- Cross-check facts against § 5.4

16.19 가드레일 (guardrails)

A term referring to mechanisms that constrain the inputs and outputs of an AI system to prevent unintended behavior or dangerous outcomes. It is implemented at several layers, including prompt filtering, output validation, and execution permission restrictions.

This is a general technical term with no origin attributable to a specific person or point in time.

In coding agents, it is implemented in forms such as restricting file access scope, blocking dangerous commands, and requiring approval before execution, and it overlaps considerably with human in the loop (G11). It is discussed in the context of security and accountability in § 9.2 and § 9.5 respectively.

ℹ Extension pointers

- Compare how guardrails are implemented across different coding agent tools (permission systems, sandboxes, etc.)
- Clarify the relationship with guardrail bypass cases (prompt injection, etc.)
- Cross-check facts against § 9.2, § 9.5

16.20 Software 2.0 / 3.0 (Software 2.0 / 3.0)

Software 2.0 refers to the paradigm of building programs by training neural network weights on data, rather than having people write rules directly as code. Software 3.0 continues this lineage and is used to refer to the stage in which a model's behavior is programmed through natural language prompts.

Software 2.0 is a concept coined by Andrej Karpathy in a piece he wrote in 2017 (link). It remains to be confirmed who formalized the follow-up concept of Software 3.0, and when.

Software 2.0 has become established as a term for deep learning in general, while Software 3.0 is used in vibe coding discourse as a concept that underpins the view of natural language as a program.

ℹ Expansion pointers

- Confirm the original source of who first formalized the concept of “Software 3.0” and when (including Karpathy’s later talks and writings)
- Confirm in the original source where the three-stage distinction of Software 1.0/2.0/3.0 first appeared, and whether 3.0 was already present in the 2.0 article
- Work out how this connects conceptually to § 8.1 (from next-token prediction to code)

17 Reading List

This is the list of materials that forms the common foundation for the course and the writing. Presenters add materials for their own chapter here. Rule: prioritize primary sources, note the date checked, and add a one-line summary.

17.1 Primary Sources

- Andrej Karpathy, The original vibe coding tweet (X, 2025-02). The birth of the term. “fully give in to the vibes, embrace exponentials, and forget that the code even exists.”
- Andrej Karpathy, One-year retrospective tweet (X, 2026-02). His own look back at how the term spread. “shower of thoughts throwaway tweet.”
- Andrej Karpathy, Software 2.0 (Medium, 2017). The origin of the discussion that neural networks replace code.

17.2 Discourse

- Simon Willison, Not all AI-assisted programming is vibe coding (but vibe coding rocks) (2025-03-19). The definitive piece distinguishing the narrow and broad senses of the term.
- Collins Dictionary, Word of the Year 2025 announcement (2025-11). Its entry into everyday vocabulary. Related coverage: CNN.
- CodeRabbit, A semantic history of vibe coding . A summary of how the term’s meaning changed over time.
- Karpathy, Tweet endorsing context engineering (2025-06). The moment that formalized Tobi Lütke’s coinage of the term (2025-06-19). Summary pieces: Simon Willison, Phil Schmid.
- GitHub, Spec-driven development with AI: Spec Kit release (2025-09). A spec-first workflow as a reaction against vibe coding.
- Simon Willison, Vibe engineering (fall 2025). An attempt to redefine it as accountable acceleration.
- ADTmag, Loop Engineering Emerges as Developers Put AI Coding Agents on Repeat (2026-07-01). Coverage of the emergence of the loop engineering concept.
- Faros AI, Harness Engineering (2026). “Agent = Model + Harness,” a three-stage theory of prompt → context → harness.

- O'Reilly Radar podcast, Vibe Coding with Steve Yegge. An interview with the person who coined CHOP (chat-oriented programming). Also see the book "Vibe Coding" by Gene Kim and Steve Yegge (add publication details once confirmed).

17.3 Practice

- Anthropic, Building effective agents (2024-12). The reference document for agent design principles.
- Anthropic, Official prompt engineering guide. The reference document for the prehistory side.

17.4 Education and Learning

- David Deming, Lucy Swedberg, The Value of Human Skills in an AI Economy (Harvard Business Impact Education webinar, 2026-07-28, checked 2026-07-29). They locate humans' comparative advantage in rapid social learning and strategic interaction, and propose that in the classroom students should wrestle with material on their own first, then use AI for critique and feedback.

> **i** Pending Collection
>
> The original post on loop engineering (Addy Osmani, 2026-06) and the practitioner writings that were its origin, Korean discussion materials, and the key literature for each chapter. Presenters should fill this in as part of their chapter research. Add links only after confirming they work.

18 Contributors

This book is written collaboratively in the Seoul National University Special Topics in Data Science course, "Vibe Coding." Each edition records that semester's contributors.

18.1 2026 Edition (2026, 2nd Semester)

Lead

- Hyunwoo Park (Seoul National University Graduate School of Data Science) · Planning and editorial lead

Teaching Team

- (to be listed once confirmed)

Authors (Students)

- (to be listed once chapters are assigned. Format: Name · Assigned chapter)

> **Note**
>
> Students add their name to this list when their first PR is merged. That is this course's rite of passage.