

Visual Guide to Computer Systems

Vibe Coding A Visual Guide

Sample Book — Chapters 1 to 2

Prompt, Context, Harness, and Loop

Reiner Braun

Introduction

This book is designed for readers who are taking their first steps with Vibe Coding, including systems engineers who want to build the right foundation from the start.

Rather than asking you to memorize more than you need, the book focuses on the essential concepts that will support your future study and day-to-day work. More advanced topics are included only when they are likely to become useful to many readers, and even then they are introduced only far enough to explain their purpose, context, and overall role.

Processes, relationships, and structures that can be difficult to picture from text alone are supported by diagrams. By moving between explanations, visuals, and examples, you can build a clearer and more practical understanding of the subject.

Product names, service names, logos, icons, and other identifying marks mentioned in this book are trademarks or registered trademarks of their respective owners. This book is not an official publication, certified training resource, or endorsed learning material of any company or organization.

How to Use This Book

Read the chapters in order on your first pass. Each chapter assumes the foundation established by the chapters before it, so this sequence builds the required background knowledge step by step.

At the beginning of each chapter, review **What You'll Be Able to Do** before reading the main explanation. These outcomes identify the practical understanding you should gain and help you distinguish essential material from supporting detail.

Use the diagrams to understand relationships, processes, boundaries, and structures that are harder to picture from prose alone. The surrounding text remains the authoritative explanation; each diagram is a visual aid for organizing that explanation.

Try the code, commands, and hands-on examples yourself whenever you have a safe local environment. Before running a command, confirm its stated prerequisites and compare your result with the expected outcome rather than copying the example mechanically.

At the end of each chapter, use the **Chapter Summary** and **Review Questions** to confirm that you can explain the fundamentals in your own words. Revisit the relevant section when an answer depends mainly on memory or guesswork.

Advanced-topic overviews do not need to be memorized at this stage. Treat them as signposts: they show what the topic is for, where it fits, and when you may need to study it in greater depth.

Table of Contents

[Introduction](#)

[How to Use This Book](#)

[The Core Conclusion](#)

[Code Generation Is a Proposal, Not Proof](#)

[What “Vibe Coding” Means in This Guide](#)

[The Four Layers at a Glance](#)

[The New Bottleneck](#)

[Good Early Uses](#)

[Uses That Require Stronger Control](#)

[Classifying a Task Before Starting](#)

[The Systems Engineer’s Continuing Role](#)

[A First Practical Decision](#)

[Chapter 2 — A Minimum Mental Model of Language Models and Coding Agents](#)

[A Coding Agent Is Not a Deterministic Programmer](#)

[Tokens and the Context Window](#)

[Four Sources of Apparent Knowledge](#)

[Why Plausible Fabrications Occur](#)

[Tools Turn a Model into an Agent](#)

[A Traced Example](#)

[No Automatic, Reliable Long-Term Memory](#)

[Self-Assessment Is Not Independent Verification](#)

[Model Capability Does Not Remove Engineering Discipline](#)

Main Text

The Core Conclusion

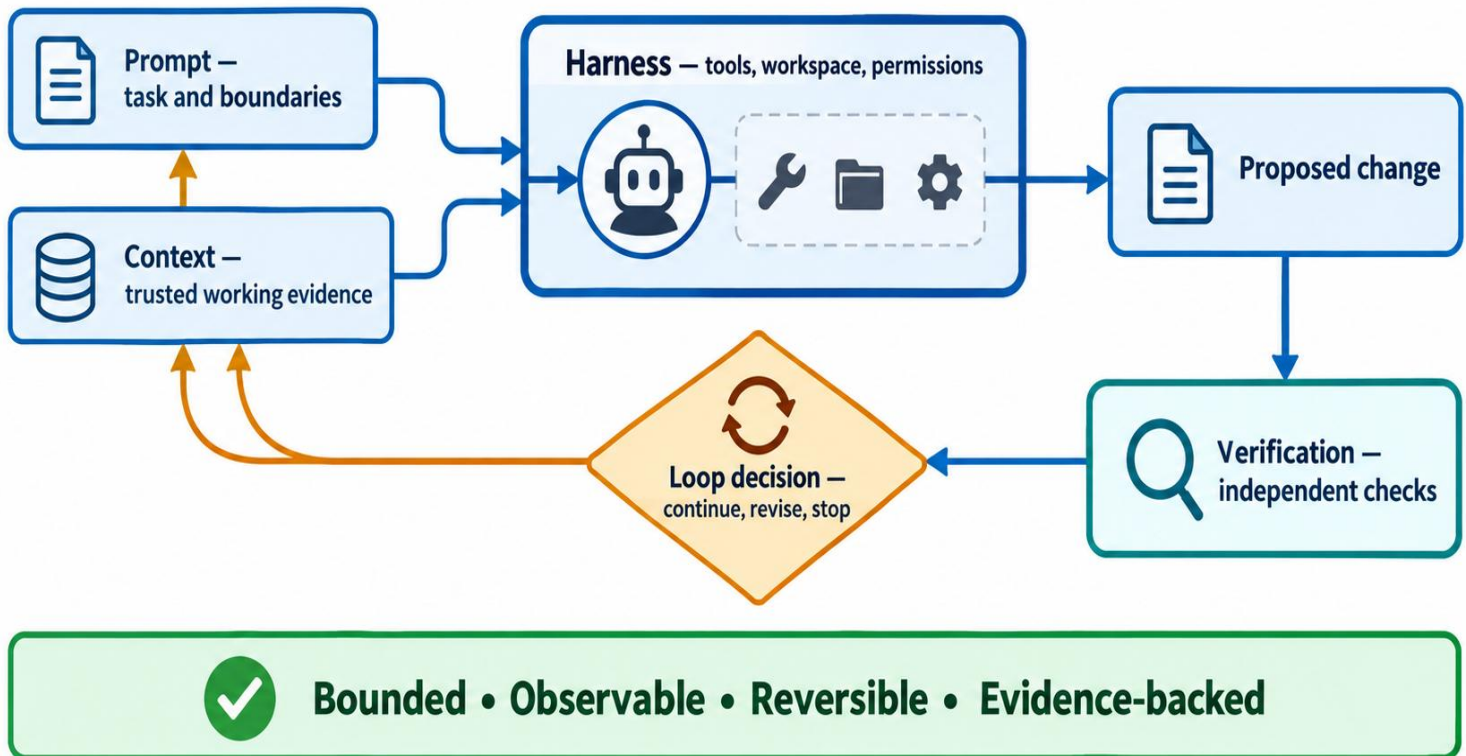
Reliable AI-assisted coding is not mainly a prompting technique. It is a controlled engineering system. A language model can produce code quickly, but speed of generation does not prove that the code is correct, safe, compatible, or maintainable. The engineer must therefore shape the work through four connected layers:

- **Prompt:** the task, boundaries, process expectations, and requested result.
- **Context:** the evidence the model can use, such as source files, tests, logs, interfaces, and decisions.
- **Harness:** the tools, workspace, permissions, and automated checks around the model.
- **Loop:** the repeated process of observing, acting, checking, and deciding what to do next.

These layers work as a system. A precise prompt cannot compensate for a missing source file. Complete context cannot compensate for unsafe permissions. Good tools cannot compensate for a weak test. A fast loop cannot compensate for an incorrect requirement.

Chapter 0: Book Opening

0-1 The Core Conclusion



001

Figure 0-1: The Core Conclusion

The practical goal is therefore not to make the model appear intelligent. The goal is to make each change **bounded, observable, reversible, and supported by evidence**.

Chapter 1 — AI Coding as a Controlled Engineering System

What You'll Be Able to Do

After completing this chapter, you will be able to:

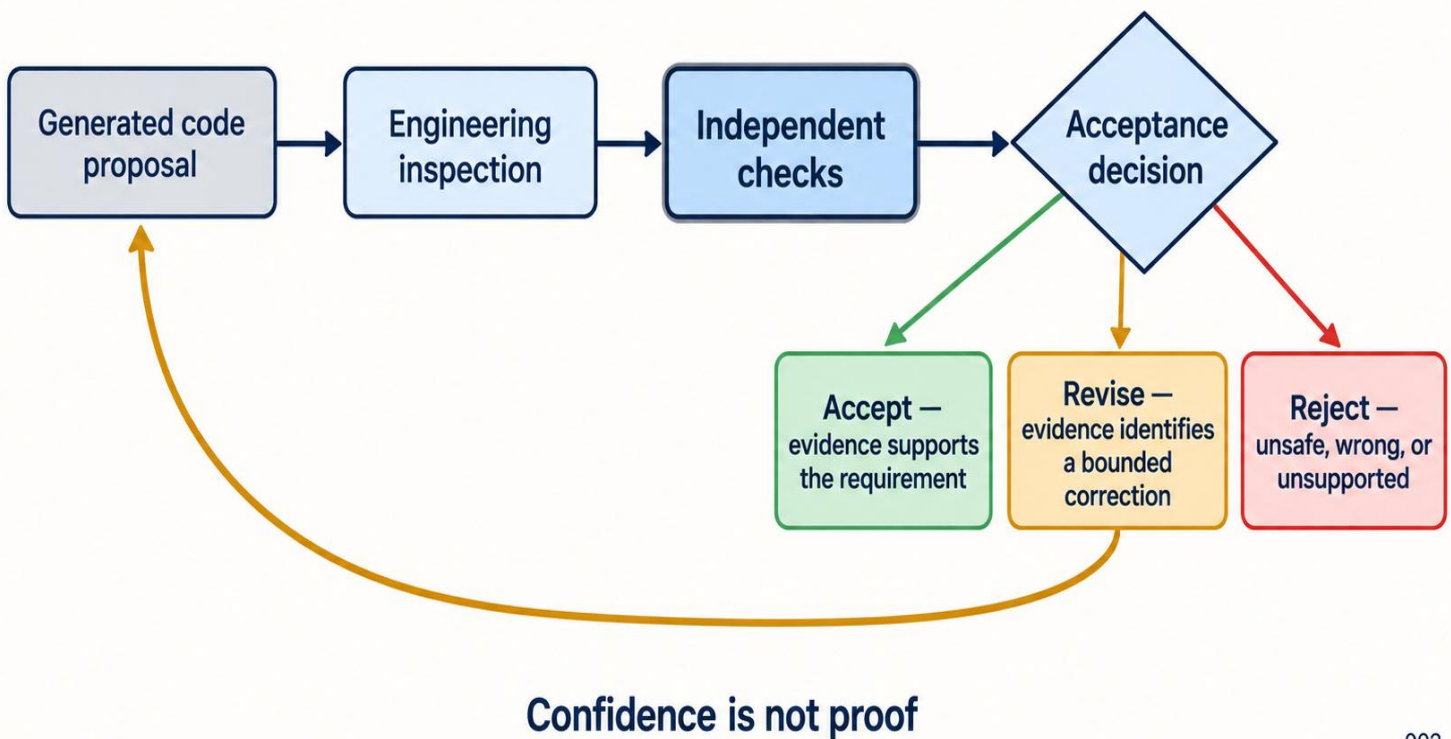
- Explain why generated code is a proposal that still requires engineering evidence.
- Describe the distinct roles of Prompt, Context, Harness, Loop, and verification.
- Classify an AI-assisted task by risk, reversibility, and verifiability before starting.

Code Generation Is a Proposal, Not Proof

The most important principle is simple: **AI-generated code is a candidate solution**. It is not evidence that the requirement has been met.

Chapter 1: AI Coding as a Controlled Engineering System

1-1 Code Generation Is a Proposal, Not Proof



002

Figure 1-1: Code Generation Is a Proposal, Not Proof

A language model may produce code that is syntactically clean, follows familiar patterns, and includes a confident explanation. None of those qualities proves that the code matches the actual repository, preserves existing behavior, handles failures correctly, or satisfies the team's operational constraints.

This distinction is familiar in ordinary engineering. A design sketch can be useful without being a completed bridge. A proposed network configuration can be plausible without being safe to apply. In the same way, generated code can accelerate implementation while still requiring inspection and verification.

The engineer remains accountable for questions such as:

- Does the change solve the stated problem?
- Does it preserve required interfaces and compatibility?

- Can it expose data, credentials, or infrastructure?
- Can it be tested independently?
- Can it be reversed if the result is wrong?
- Will operators and maintainers understand its behavior?

The model can help investigate and implement. **Responsibility for accepting the result remains with the engineering process and the people who own the system.**

What “Vibe Coding” Means in This Guide

The phrase **vibe coding** is commonly used for software development in which a person describes an intended result in natural language and lets an AI system produce much of the implementation. In its most informal form, the person repeatedly asks for changes until the software seems to work, without closely understanding the code.

That informal approach can be useful for disposable experiments. It becomes dangerous when applied unchanged to systems that have users, stored data, operational dependencies, security requirements, or maintenance obligations.

This guide uses the phrase in a disciplined setting. The objective is not blind acceptance of generated code. It is **AI-assisted engineering in which natural-language direction is combined with focused context, controlled tools, independent verification, and explicit stopping rules.**

The difference is not whether AI writes code. The difference is how the work is controlled.

Informal experimentation	Professional AI-assisted engineering
Success means the output appears to work.	Success means defined acceptance criteria are supported by evidence.
The model may change anything convenient.	Scope, interfaces, and prohibited changes are explicit.
Errors lead to another broad request.	Errors are classified and used to choose a bounded next action.
The same workspace may contain unrelated data and credentials.	The workspace and permissions are limited to the task.
The conversation acts as the project record.	Durable engineering artifacts record requirements, changes, and results.

The left-hand approach is not always wrong. It is appropriate when the cost of failure is low, the work is disposable, and no sensitive resource is exposed. The error is treating it as suitable for every task.

The Four Layers at a Glance

A practical analogy is a technician completing a repair in a workshop.

The **prompt** is the work order. It states what is wrong, what outcome is required, and which constraints matter.

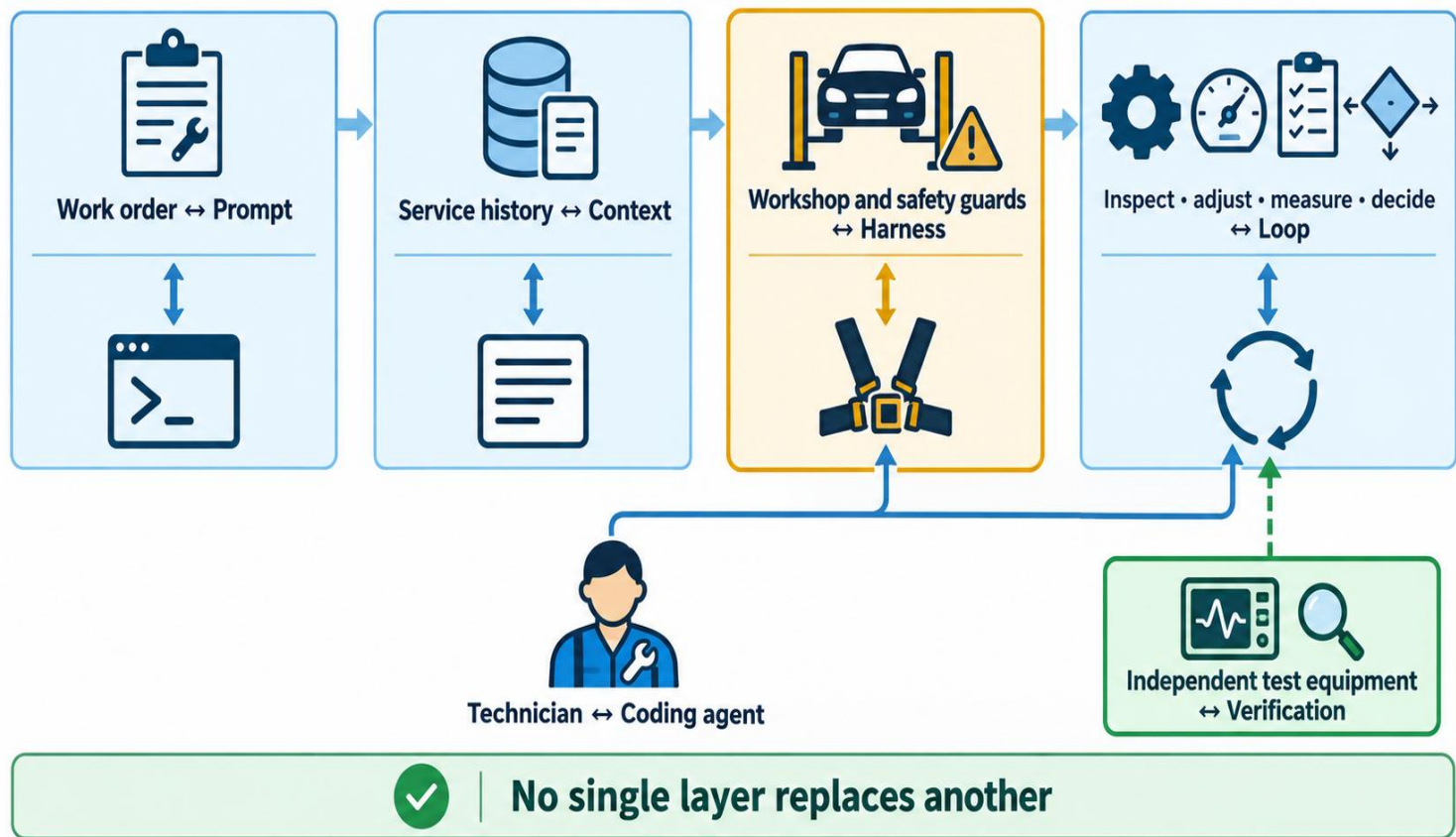
The **context** is the service history, diagrams, measurements, known faults, and parts documentation. A good work order is not enough if the technician lacks the correct model information.

The **harness** is the workshop: tools, safety guards, test equipment, access controls, and a clean place to work. Even a skilled technician is limited by poor instruments or unsafe conditions.

The **loop** is the repeated cycle of inspect, adjust, measure, and decide. A single attempt is rarely the whole repair. Each measurement changes the next action.

Chapter 1: AI Coding as a Controlled Engineering System

1-3 The Four Layers at a Glance



003

Figure 1-2: The Four Layers at a Glance

The analogy also shows why the layers cannot be reduced to one “perfect prompt.” A longer work order cannot replace a missing diagnostic tool. More service records cannot replace a safety guard. Repeating the same repair cannot replace a measurement that reveals the real fault.

The four layers answer different questions:

Layer	Main question	Typical failure
Prompt	What should happen next, and within what boundaries?	The task is vague, contradictory, or too broad.
Context	What evidence does the model need for the next decision?	Relevant information is absent, stale, misleading, or buried.
Harness	What can the agent access, change, execute, and check?	Tools are missing, permissions are excessive, or results are unclear.
Loop	How does evidence determine the next action and the stopping point?	The agent retries blindly, expands scope, or stops without proof.

A fifth term will also appear throughout the guide: **verification**. Verification is the use of independent, repeatable checks to determine whether a requirement or invariant has been satisfied. It is part of the harness and provides the evidence that drives the loop.

The New Bottleneck

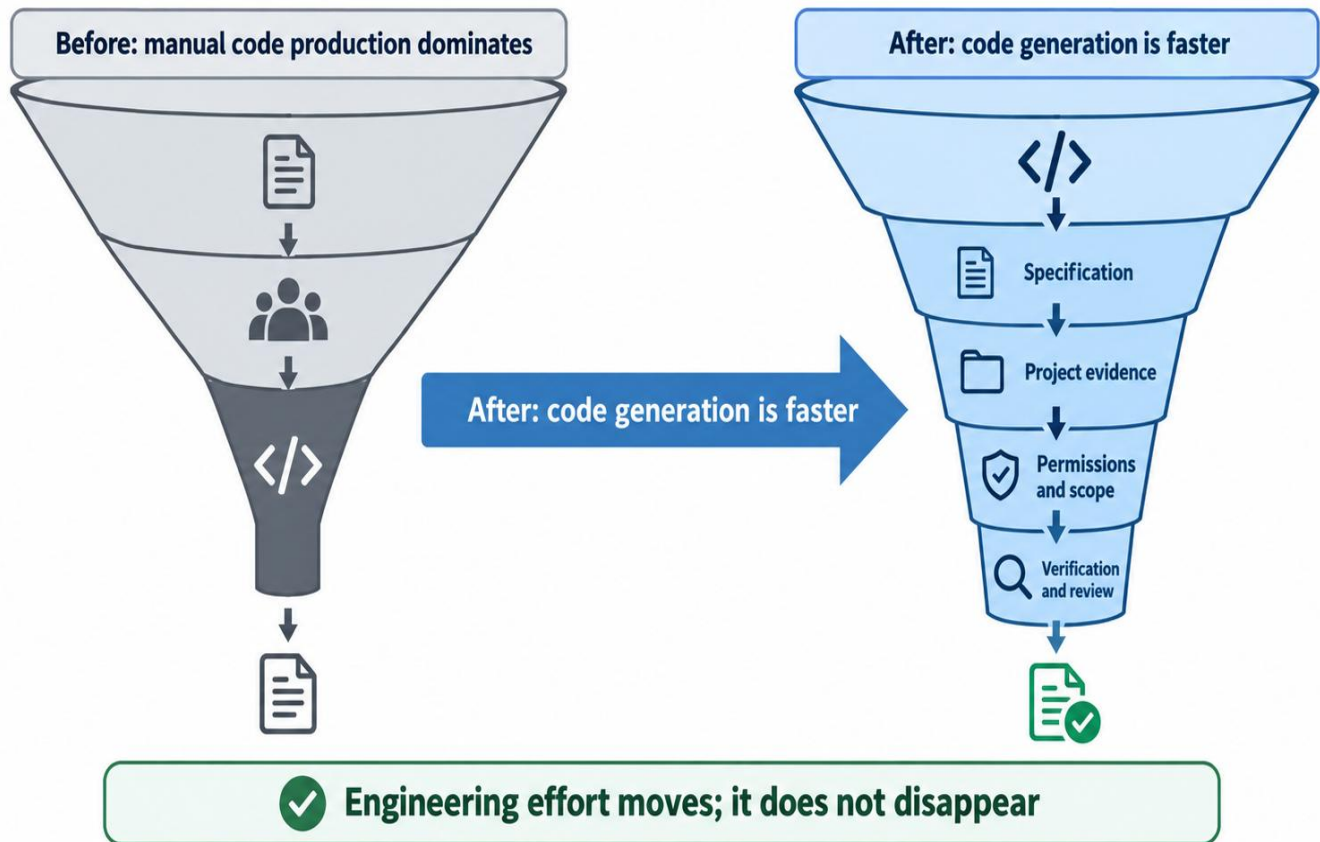
Traditional programming often makes implementation effort visible: an engineer must type code, search documentation, connect components, and correct syntax. AI tools reduce some of that effort. As code generation becomes faster, the bottleneck moves.

The difficult work becomes:

- defining the intended behavior precisely;
- locating the authoritative project evidence;
- restricting what the agent may change;
- constructing checks that can distinguish success from plausible failure;
- reviewing the effect on the surrounding system;
- deciding when to stop, revert, or escalate.

Chapter 1: AI Coding as a Controlled Engineering System

1-4 The New Bottleneck



004

Figure 1-3: The New Bottleneck

This is why a systems engineer is well positioned to use coding agents effectively. Systems engineering already emphasizes boundaries, interfaces, failure modes, feedback, observability, and operational control. The same habits that make services reliable also make AI-assisted coding reliable.

Good Early Uses

Beginners should start with tasks that have a small **blast radius**, meaning a limited range of possible damage if the change is wrong. The best early tasks are also reversible and easy to verify.

Suitable examples include:

- explaining a small, already available section of a repository;
- locating where a configuration value is read;
- adding a focused regression test for a known defect;
- fixing a small validation error with stable acceptance criteria;
- performing a repetitive refactor protected by existing tests;
- updating documentation after confirming the actual code behavior;
- generating a draft script in an isolated environment, followed by review and testing.

A **regression test** is a test added to ensure that a previously observed defect does not return. Such tasks are valuable because the defect supplies a concrete failure, and the test can become lasting evidence.

Uses That Require Stronger Control

Some tasks should not be the starting point for a beginner. They may still use AI assistance, but only inside stronger organizational and technical controls.

Examples include:

- direct production changes;
- database schema or data migrations;
- scripts with broad cloud or administrator permissions;
- authentication, authorization, cryptography, or secret-management changes;
- safety-critical decisions;
- handling confidential customer or employee data;
- large architectural rewrites;
- changes whose correctness cannot be observed or tested.

The concern is not that the model will always fail. The concern is that the **consequences of an undetected failure are too high**.

Classifying a Task Before Starting

A simple three-factor classification helps determine how much control is necessary.

1. **Risk:** What harm could occur if the result is wrong?
2. **Reversibility:** How easily can the change be undone without lasting effects?
3. **Verifiability:** How easily can correct behavior be demonstrated independently?

Chapter 1: AI Coding as a Controlled Engineering System

1-7 Classifying a Task Before Starting

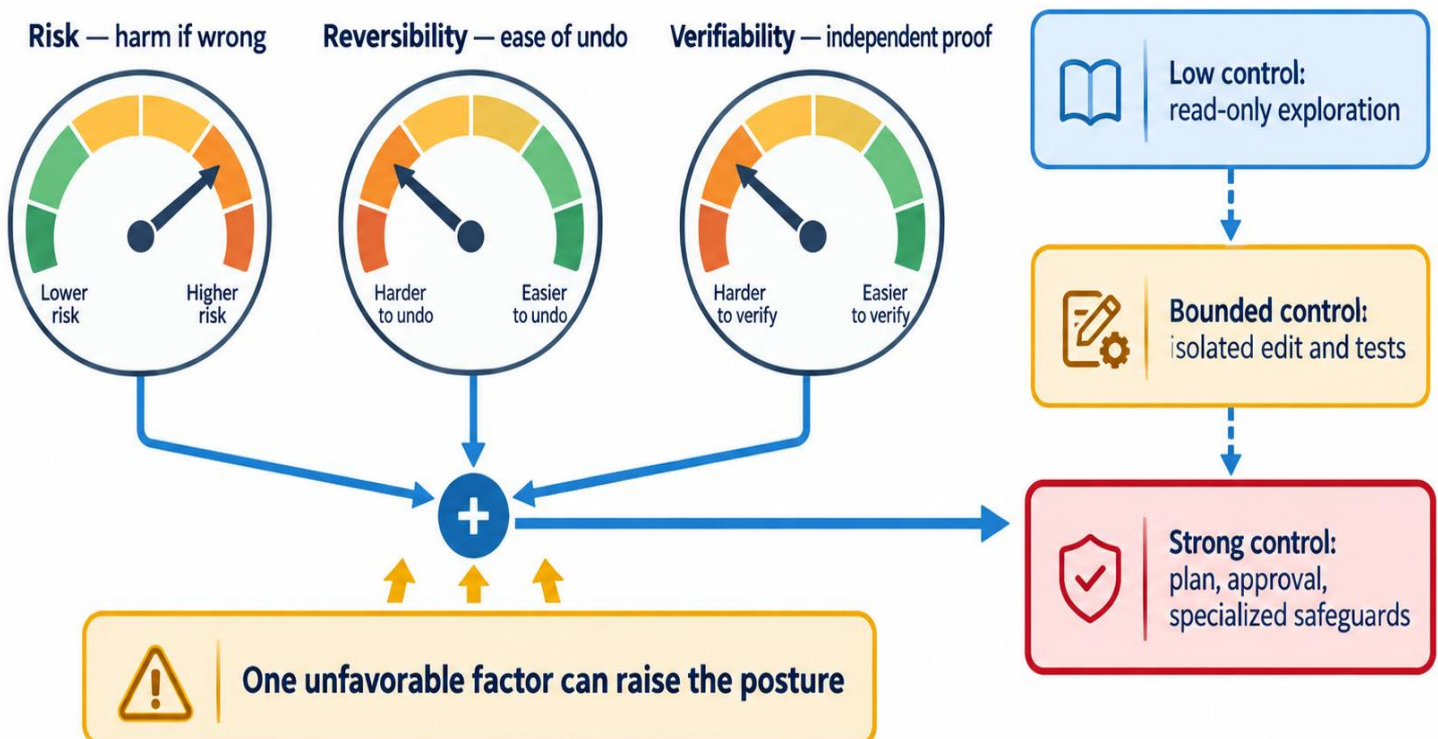


Figure 1-4: Classifying a Task Before Starting

These factors are more useful than asking whether a task “looks easy.” A one-line access-control change can be riskier than a hundred-line internal refactor.

Example task	Risk	Reversibility	Verifiability	Appropriate starting posture
Explain how a parser works without editing files	Low	High	High	Normal read-only exploration
Reject whitespace-only names and add a test	Low to medium	High	High	Bounded edit in an isolated workspace
Rename an internal helper across a tested module	Medium	High	Medium to high	Small steps with targeted and broad tests
Upgrade a core dependency across several services	Medium to high	Medium	Medium	Planned work with compatibility checks and human review
Modify production firewall rules	High	Low	Medium	Existing operational change controls and explicit approval
Run a generated migration against customer data	Very high	Low	Difficult	Do not use an informal agent loop; require specialized controls

The table does not prescribe universal ratings. The same action can have different consequences in different environments. The important habit is to make the assessment explicit.

The Systems Engineer’s Continuing Role

AI assistance does not remove systems responsibilities. It makes them more important because implementation can advance before understanding catches up.

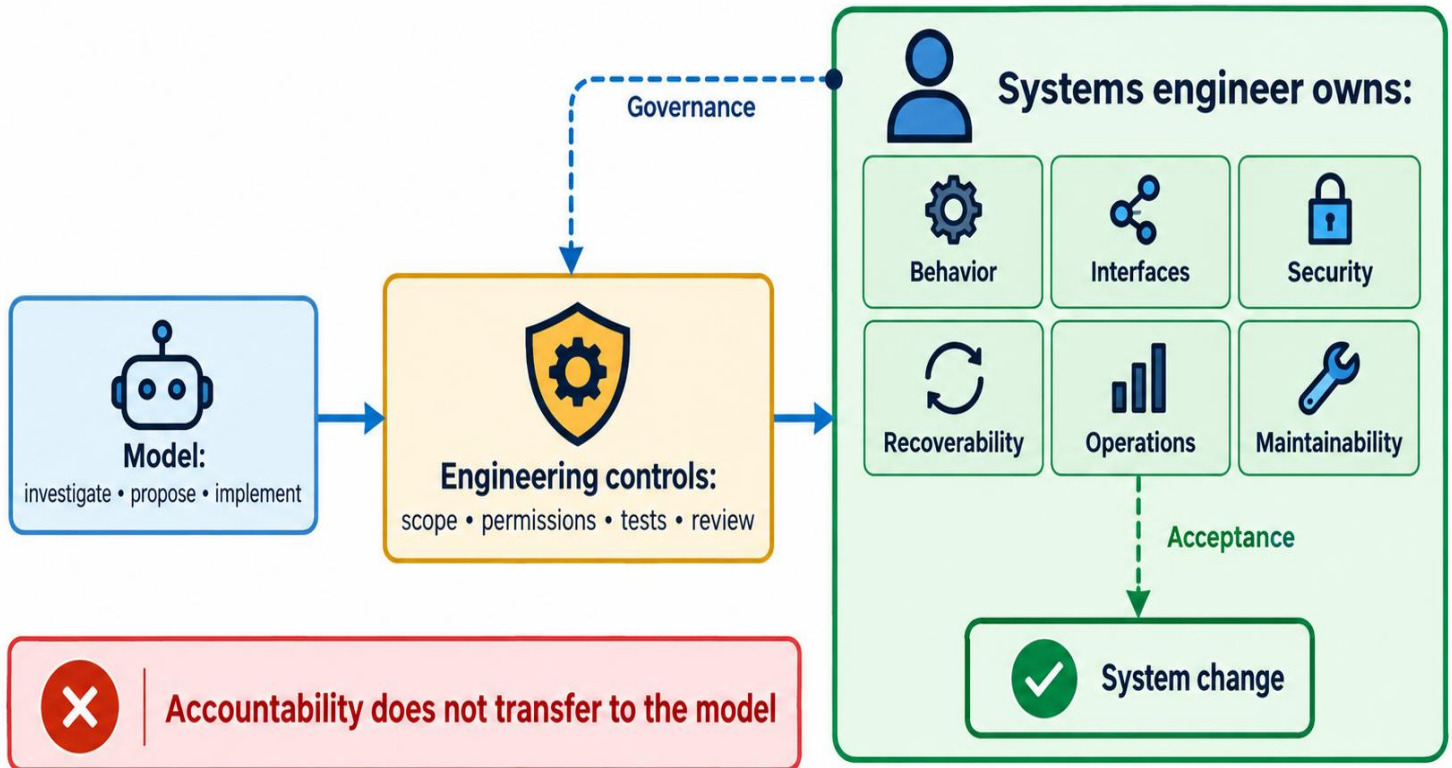
A systems engineer must continue to protect:

- **Behavior:** the observable result for users and dependent systems.
- **Interfaces:** APIs, schemas, command formats, files, and contracts used by other components.
- **Security:** trust boundaries, permissions, secrets, and untrusted input.
- **Recoverability:** the ability to restore a known-good state.
- **Operations:** logging, monitoring, resource use, deployment behavior, and failure handling.

- **Maintainability:** clear code, appropriate scope, and understandable decisions.

Chapter 1: AI Coding as a Controlled Engineering System

1-8 The Systems Engineer's Continuing Role



006

Figure 1-5: The Systems Engineer's Continuing Role

The right mental model is not “the AI is the engineer.” It is **the AI is a fast, probabilistic contributor inside an engineering control system.**

A First Practical Decision

Consider a new team member who receives this ticket:

Fix the display-name validation defect in the profile service.

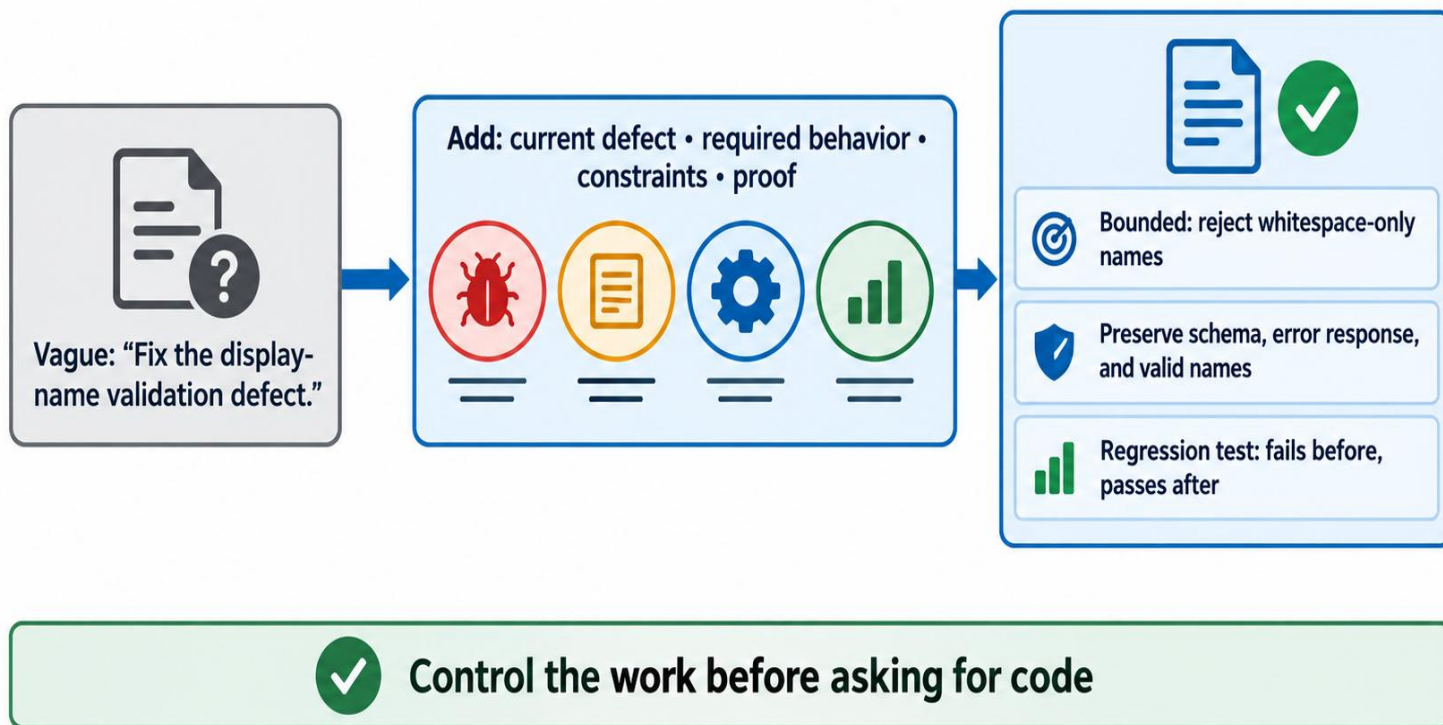
The request is not yet safe to implement. It does not state the current defect, required behavior, affected interface, compatibility expectation, or proof of completion.

The task becomes a good beginner assignment only after those elements are made observable. For example:

The profile service currently accepts a display name containing only whitespace. Reject whitespace-only values with the existing validation error response. Do not change the public request schema or the behavior for valid names. Add a regression test that fails before the fix and passes afterward.

Chapter 1: AI Coding as a Controlled Engineering System

1-9 A First Practical Decision



007

Figure 1-6: A First Practical Decision

This version does not dictate the implementation. It defines a small result that can be inspected, tested, and reverted.

The first discipline of vibe coding is therefore not asking for code. It is deciding whether the work can be controlled.

Chapter Summary

Reliable AI-assisted coding is a controlled engineering system, not a shortcut around accountability. Begin with small, reversible, verifiable tasks and increase control as potential harm or uncertainty increases.

Review Questions

1. Why is fluent, syntactically valid code not proof that a requirement has been met?
2. What different responsibility does each of Prompt, Context, Harness, and Loop carry?
3. Which task property would make you increase controls even when the code change is small?

Chapter 2 — A Minimum Mental Model of Language Models and Coding Agents

What You'll Be Able to Do

After completing this chapter, you will be able to:

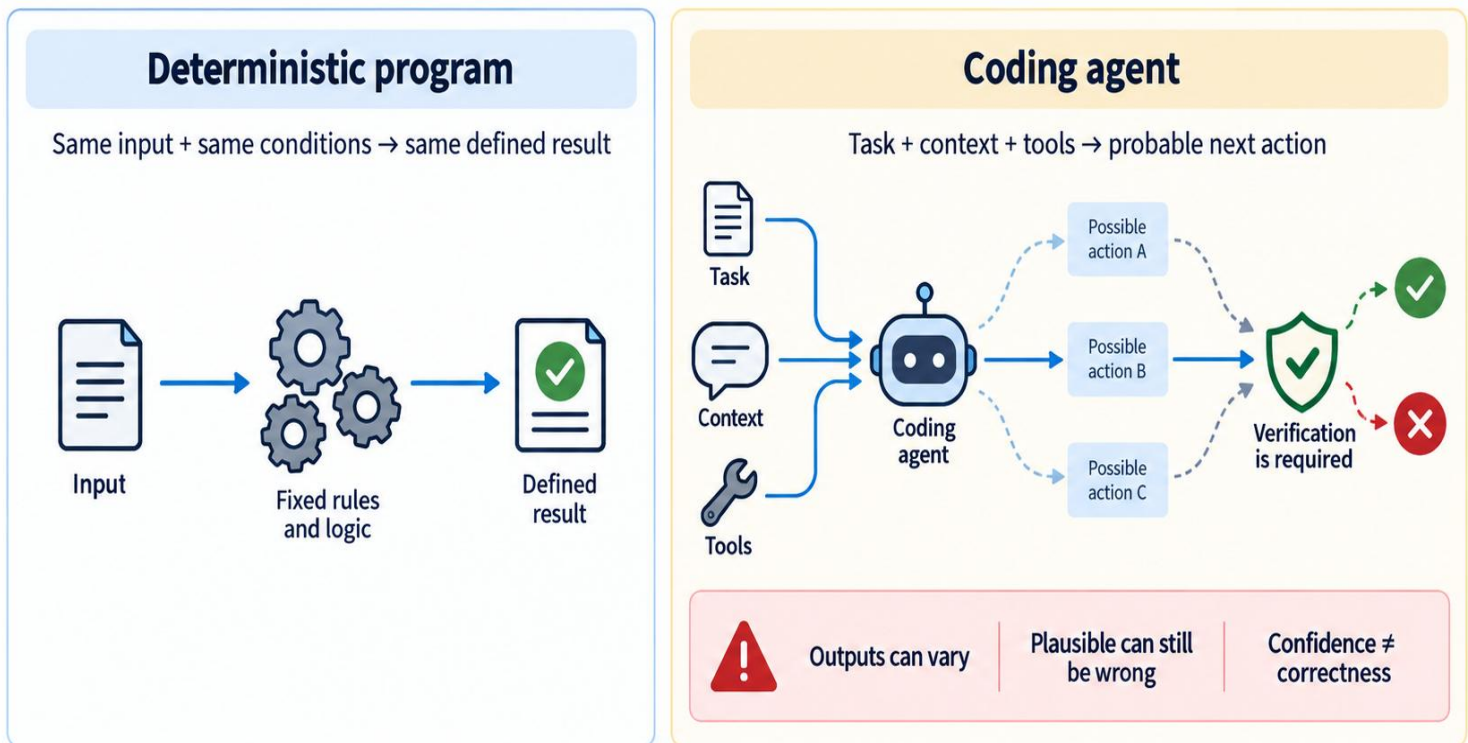
- Explain the minimum language-model concepts needed to supervise a coding agent.
- Distinguish model knowledge, supplied context, retrieved information, and tool results.
- Treat agent explanations and self-assessments as clues rather than independent proof.

A Coding Agent Is Not a Deterministic Programmer

A coding agent is best understood as a **probabilistic language model connected to tools**. It can analyze text, propose actions, inspect files, execute commands, and revise its work, but it does not automatically possess complete or reliable knowledge of the project.

Chapter 2: A Minimum Mental Model of Language Models and Coding Agents

2-1 A Coding Agent Is Not a Deterministic Programmer



008

Figure 2-1: A Coding Agent Is Not a Deterministic Programmer

A deterministic program produces the same result from the same input under the same conditions. A language model is probabilistic: it generates likely continuations based on patterns learned during training and the information currently supplied. Its output can vary, and a plausible answer can still be false.

This explains several behaviors that surprise beginners:

- the model can sound certain while being wrong;
- it can invent a file, command, option, or library function that would be reasonable but does not exist;
- it can overlook a constraint mentioned much earlier;
- it can solve a familiar-looking problem instead of the repository's actual problem;

- it can produce different implementations for the same request.

These are not unusual exceptions. They follow from how the system works.

Tokens and the Context Window

A **large language model**, usually shortened to **LLM**, is a model that processes and generates sequences of text units called tokens. A **token** is a small unit of text, often a word fragment, punctuation mark, or short word.

The model does not read a repository in the same way a person opens an entire project in an editor. It processes the information placed in its current **context window**, which is the limited amount of instructions, conversation, file content, and tool output available during a sequence of model decisions.

The context window has two practical consequences.

First, information outside the window is not automatically available. A file that exists in the repository but has not been retrieved is not reliable working knowledge.

Second, filling the window with more text does not guarantee better reasoning. Irrelevant logs, duplicated instructions, stale design notes, and complete files that are not needed can compete with the important evidence.

An everyday analogy is a workbench. A larger workbench can hold more tools and documents, but covering every surface with unrelated material makes the current repair harder, not easier.

Chapter 2: A Minimum Mental Model of Language Models and Coding Agents

2-2 Tokens and the Context Window

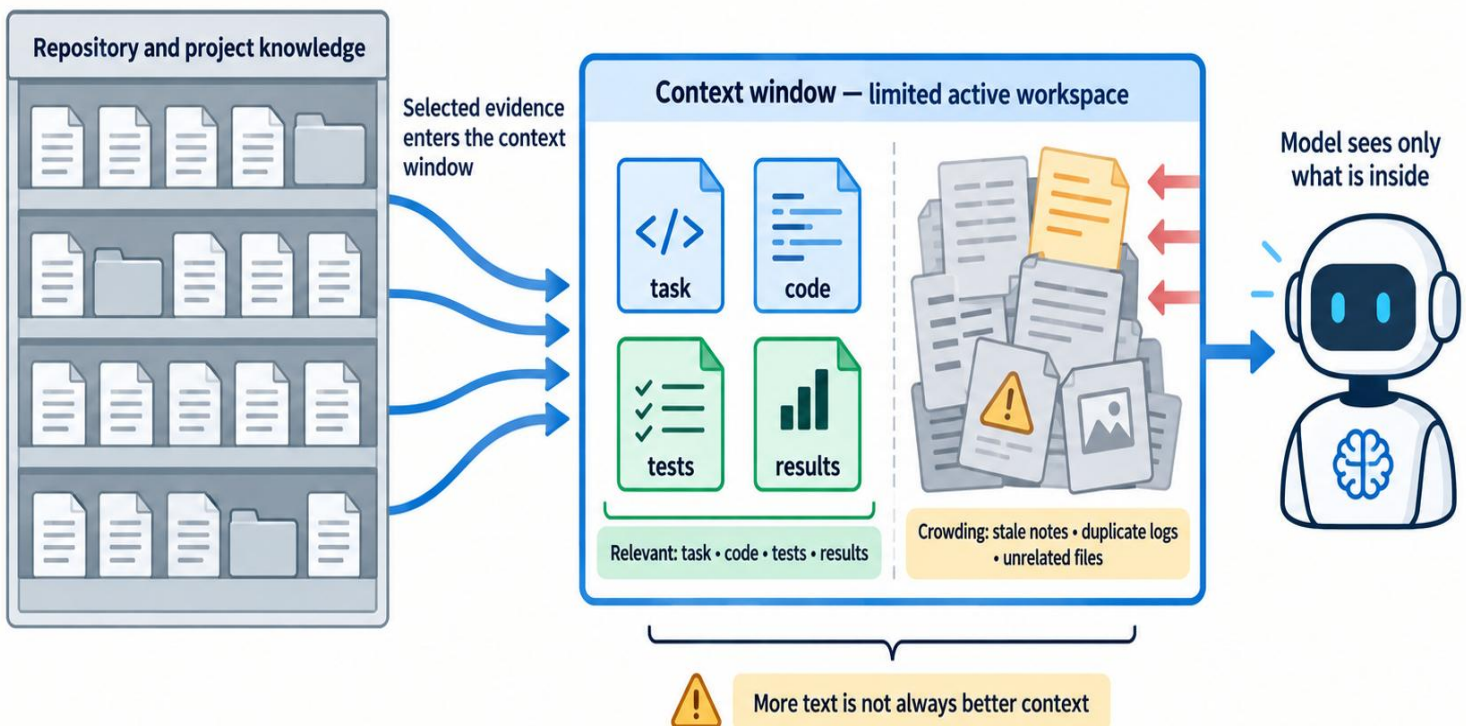


Figure 2-2: Tokens and the Context Window

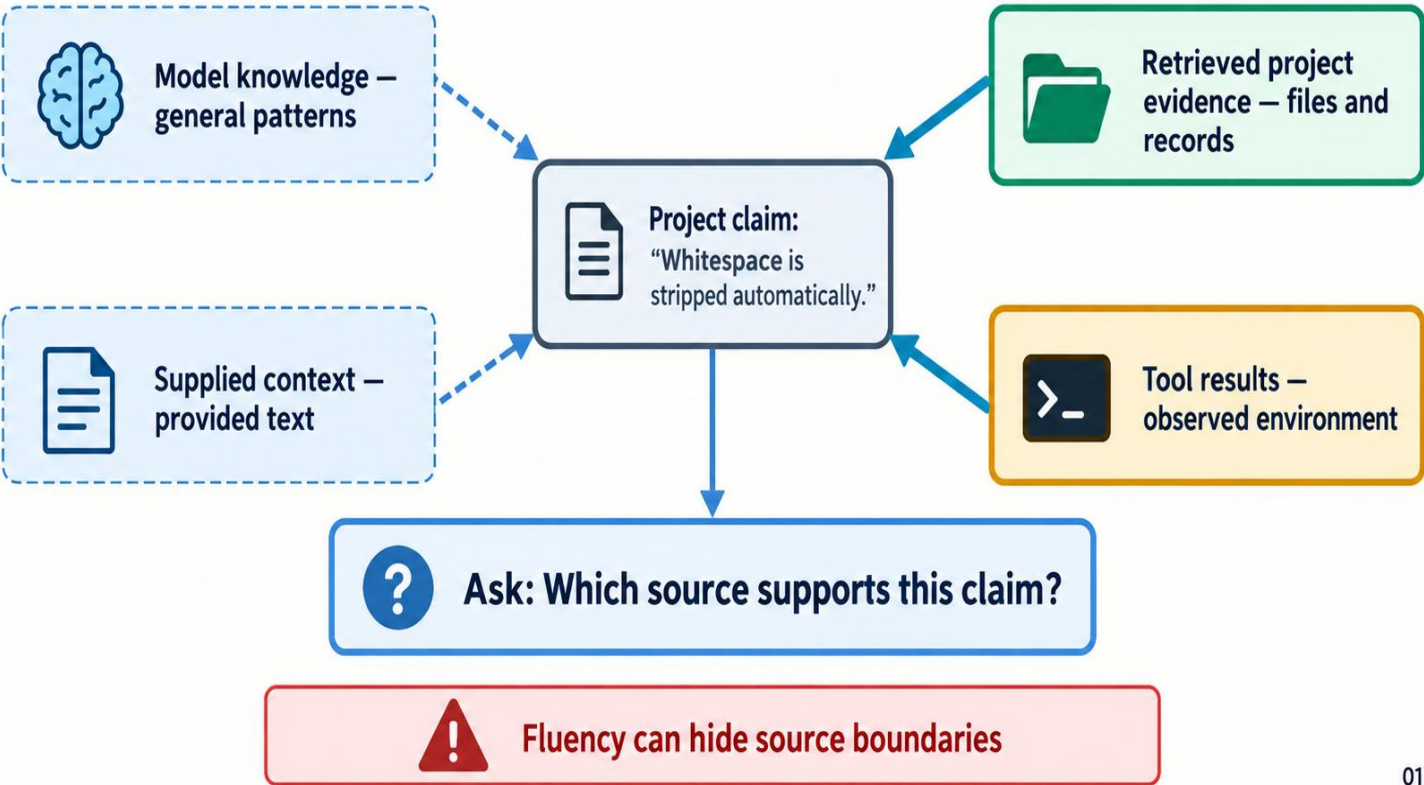
Four Sources of Apparent Knowledge

When an agent answers a project question, its response may combine several sources. They should not be treated as equally authoritative.

Source	Meaning	Reliability for the current project
Model knowledge	General patterns learned before the current task	Useful for common concepts, but may be outdated or mismatched
Supplied context	Text placed in the current interaction	Useful when relevant, current, and trustworthy
Retrieved project information	Files, search results, records, or documentation obtained during the task	Often strong evidence when the source is authoritative
Tool results	Command output, test results, compiler errors, or API responses	Strong evidence about the observed environment, but still requires interpretation

Chapter 2: A Minimum Mental Model of Language Models and Coding Agents

2-3 Four Sources of Apparent Knowledge



010

Figure 2-3: Four Sources of Apparent Knowledge

Suppose an agent says, “This framework automatically strips whitespace.” That statement could come from general model knowledge rather than the project’s installed version or configuration. A more reliable process is to inspect the dependency version, find the relevant code or documentation available to the project, and run a focused test.

Fluency hides source boundaries. The engineer must ask what evidence supports a claim.

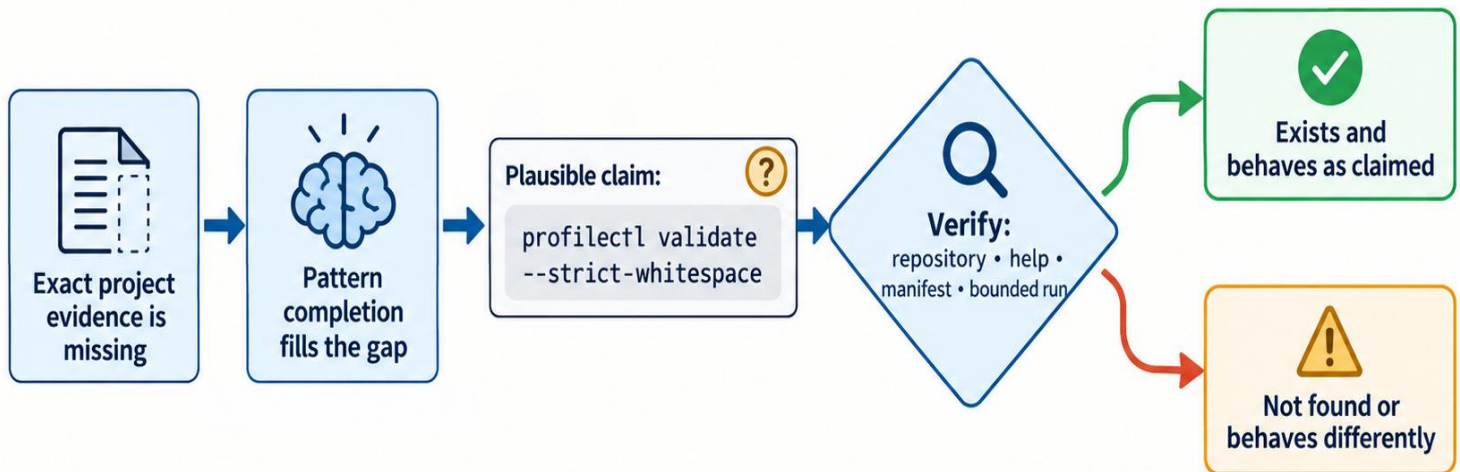
Why Plausible Fabrications Occur

A model generates text that fits the preceding context. When exact information is missing, it may produce a completion that resembles common software patterns. This can result in a plausible but nonexistent API, path, configuration key, or command.

For example, a model might suggest:

Chapter 2: A Minimum Mental Model of Language Models and Coding Agents

2-4 Why Plausible Fabrications Occur



Project claims need project evidence

011

Figure 2-4: Why Plausible Fabrications Occur

The command looks credible. That does not mean the project contains a `profilectl` program or a `--strict-whitespace` option.

The appropriate response is not to argue with the model's confidence. It is to require evidence:

- search the repository for the command;
- inspect the actual command help;
- read the relevant manifest or documentation;
- run the command in a bounded environment when safe.

A useful rule is: **the more project-specific a claim is, the more it should be grounded in project-specific evidence.**

Tools Turn a Model into an Agent

A **tool call** is a structured request from the model to an external capability, such as reading a file, searching text, running a command, editing code, or querying another service.

An **agent** is a model-based system that can choose actions, observe tool results, and continue toward a goal. The agent is not a separate kind of intelligence. It is a model operating inside an action-observation loop.

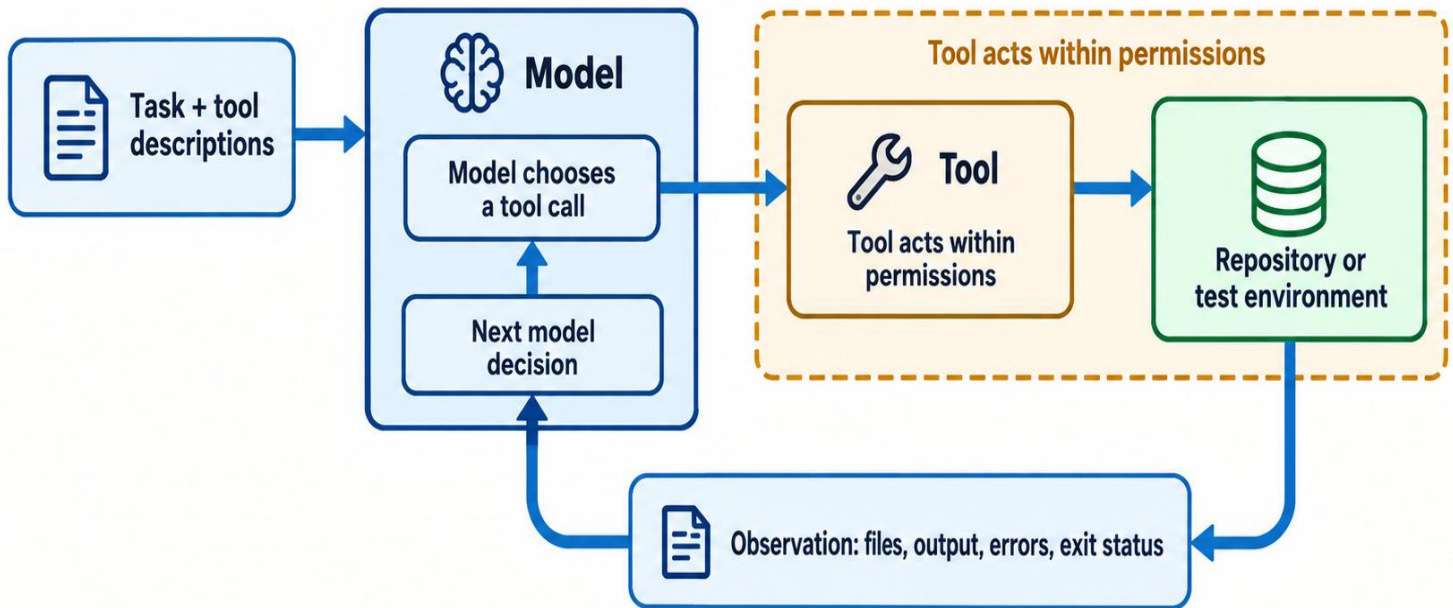
A small coding-agent sequence may look like this:

1. The agent receives the task and available tool descriptions.
2. It decides to search for display-name validation.
3. A search tool returns matching files and line locations.
4. The agent reads the implementation and relevant tests.
5. It proposes or applies a focused edit.
6. A test tool returns a failure or success result.

7. The agent interprets that result and decides whether another action is necessary.

Chapter 2: A Minimum Mental Model of Language Models and Coding Agents

2-5 Tools Turn a Model into an Agent



Agent = model + tools + action–observation loop

Harness quality shapes every action

012

Figure 2-5: Tools Turn a Model into an Agent

The quality of this sequence depends on more than the model. Search results must be relevant. File access must be correct. Edit operations must be bounded. Test output must be understandable. Permissions must be appropriate.

A Traced Example

Assume the task is to reject display names that contain only whitespace.

The initial input may contain:

- the ticket statement;
- a request not to change the public API;
- a repository map;
- the names of available tools.

The agent searches for `display_name` and finds a validation function:

```
def validate_display_name(value: str) -> None:
    if not value:
        raise ValueError("display_name must not be blank")
```

The agent notices that a non-empty string containing spaces passes the condition. It reads the existing tests and finds tests for an empty string and a normal name, but none for whitespace-only input.

It then proposes a new test and a small implementation change. The test runner produces a concrete result. If the test still fails, that result becomes new context for the next model turn.

Several important boundaries appear in this simple sequence:

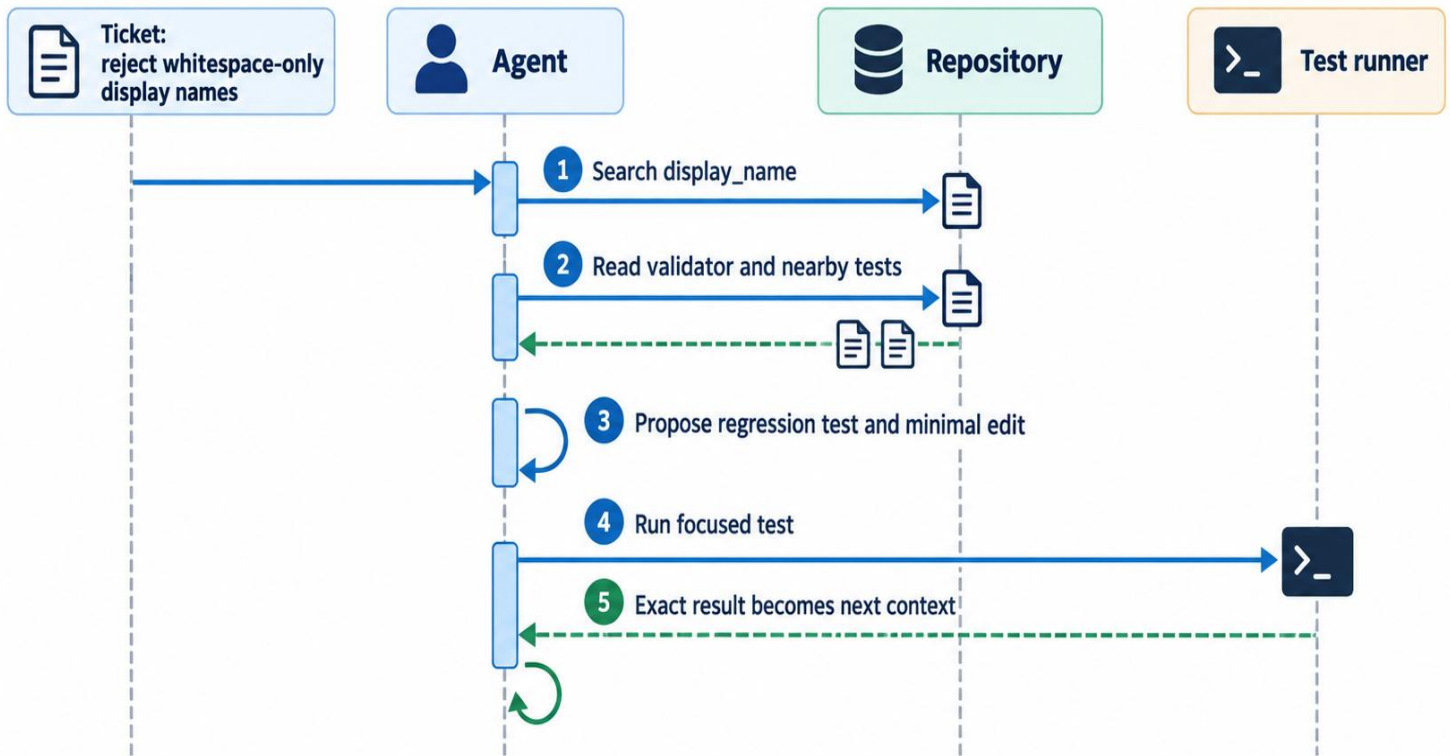
- The model did not know the implementation until it read the file.
- The search result narrowed the context.

- The existing error message constrained the change.
- The test result provided stronger evidence than the model's explanation.
- The next action depended on observed output, not on the original request alone.

This is the basic architecture of agent-assisted coding.

Chapter 2: A Minimum Mental Model of Language Models and Coding Agents

2-6 A Traced Example



The model learns the project by observing it

013

Figure 2-6: A Traced Example

No Automatic, Reliable Long-Term Memory

A beginner may assume that an agent remembers everything discussed in previous sessions. That assumption is unsafe. Some tools provide saved history or external memory, but the model itself should not be treated as having complete, durable project memory. Information may be absent from a new session, summarized incorrectly, or no longer fit inside the active context.

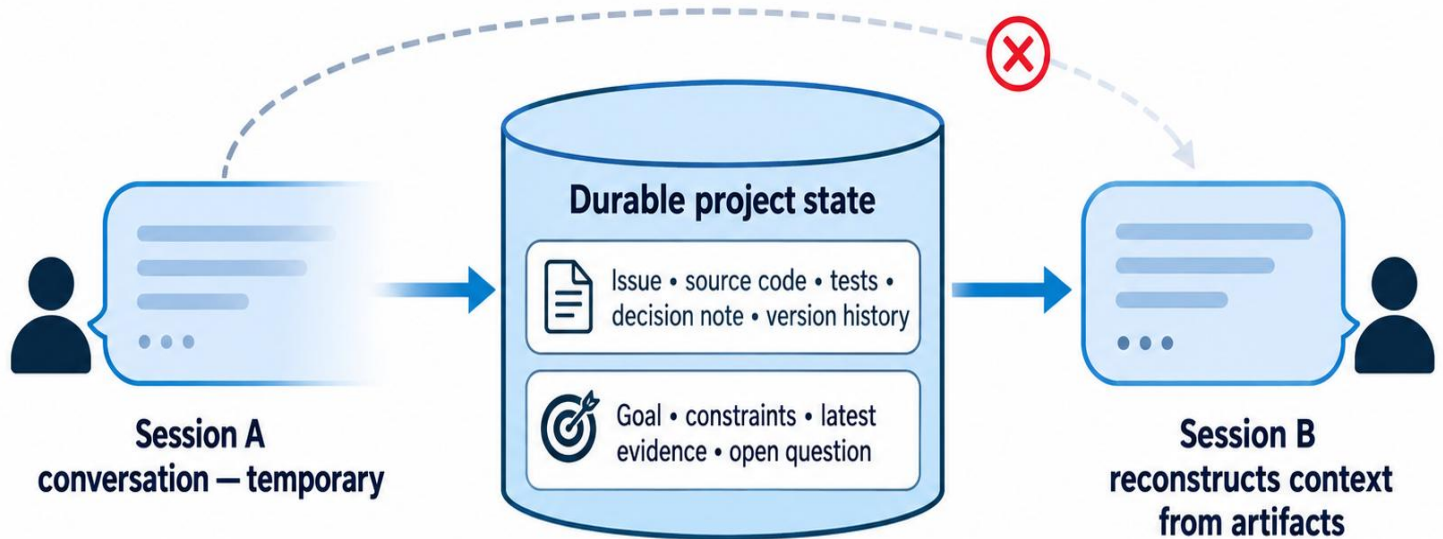
Important state should therefore exist in durable engineering artifacts, such as requirements, source code, tests, decision records, issue notes, and version-control history. A conversation can help create those artifacts, but it should not be the only place where critical decisions exist.

For a short task, a concise state note may be enough:

Goal: reject whitespace-only display names.
Constraint: preserve the request schema and existing error response.
Observed code: validation.py checks only whether the string is empty.
Evidence: new regression test fails against the baseline.
Open question: none.

Chapter 2: A Minimum Mental Model of Language Models and Coding Agents

2-7 No Automatic, Reliable Long-Term Memory



Do not rely on chat memory as the system of record

014

Figure 2-7: No Automatic, Reliable Long-Term Memory

This note is useful because it separates current facts from a long conversation.

Self-Assessment Is Not Independent Verification

An agent can explain its reasoning, summarize a diff, or state that a task is complete. These outputs are useful for navigation, but they come from the same system that produced the change.

If a student writes an answer and then marks the same answer, the result is not independent evidence. Similarly, “I verified the implementation” is weak unless the statement points to an actual test, build, static check, reproduced request, or human inspection.

The following forms of evidence are progressively stronger:

1. The agent states that the code should work.
2. The agent explains why it expects the code to work.
3. A focused test passes in the relevant environment.
4. Broader checks pass and the diff is inspected against the requirement.

5. The behavior is confirmed at the appropriate system boundary.

Chapter 2: A Minimum Mental Model of Language Models and Coding Agents

2-8 Self-Assessment Is Not Independent Verification

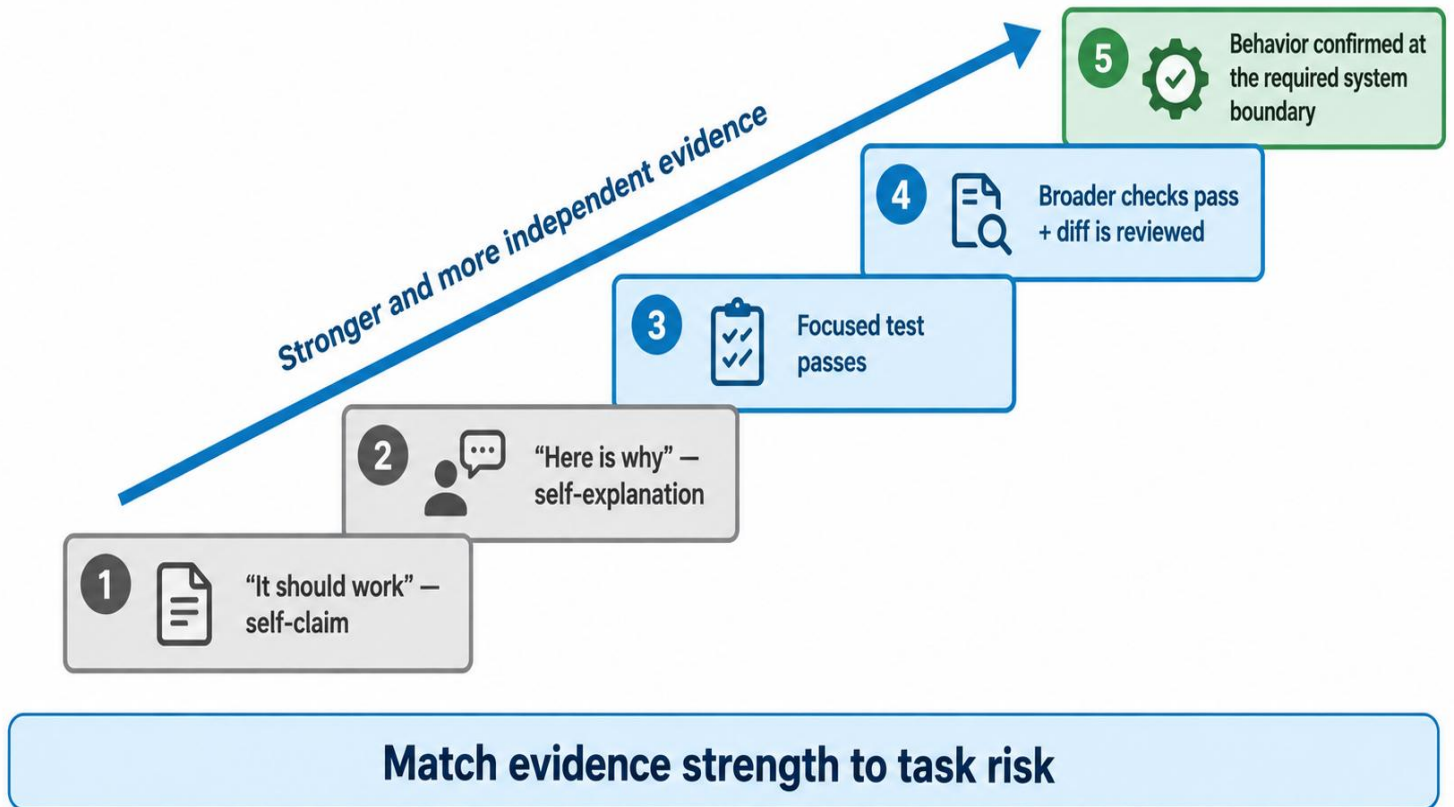


Figure 2-8: Self-Assessment Is Not Independent Verification

015

Not every task requires the highest level, but the evidence should match the risk.

Model Capability Does Not Remove Engineering Discipline

A stronger model may reason better, use tools more effectively, and make fewer mistakes. It still operates from the available instructions, context, tools, and feedback.

A model cannot reliably infer an unstated business rule. It cannot inspect a file it has not been allowed to read. It cannot prove behavior when no meaningful check exists. It cannot make broad permissions safe merely by being capable.

Model choice affects performance; system design determines control.

The minimum mental model is therefore:

- generated output is probabilistic;
- project facts must be supplied or retrieved;
- tool results are observations, not automatically correct interpretations;
- memory must be made durable outside the active conversation;
- verification must be independent enough to catch plausible mistakes.

Chapter Summary

A coding agent is a probabilistic language model connected to tools. Its useful behavior depends on the available context, tool observations, and control process; it does not possess complete project knowledge or reliable automatic memory.

Review Questions

1. How does a coding agent differ from a deterministic program?
2. Why can an agent confidently mention a file or API that does not exist?
3. What evidence is stronger than the agent's own statement that its change is correct?