



Software Development with Agentic Coding Tools

Agents and Their Failure Modes

Zachary G. Ives & Harry Smith
University of Pennsylvania
Fall 2026

The Agent Wrote 200 Lines That Don't Compile!

You asked the agent to check user permissions. It produced 200 lines of code, two modified entity definitions, a new ORM migration file, and a rewritten test suite.

It doesn't compile.

- You ask it to fix the error. It fixes that one — and introduces a new error two files away.
- You ask again. Third file, new error. Oh dear.

Big Question

How can we predict the relative strengths and weaknesses of an LLM agent? And how can we best use context to avoid these weaknesses?

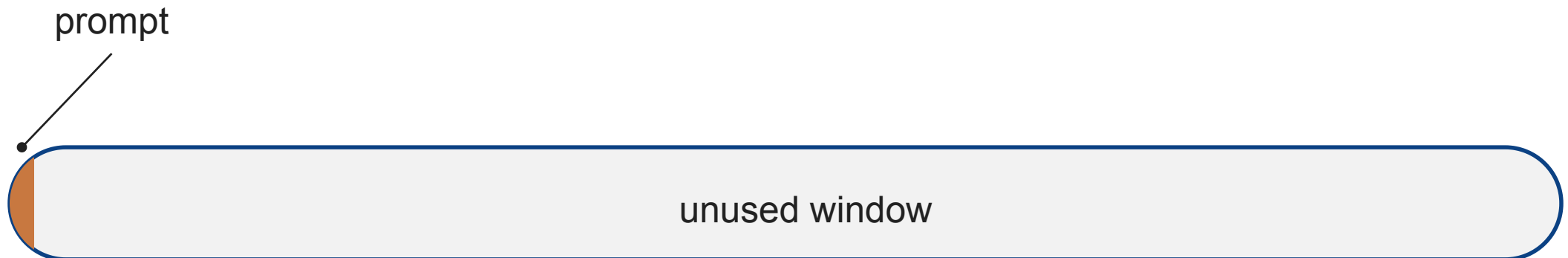
To answer this well, you need a mental model of what the agent is doing when it generates code. Then the failures make sense as natural consequences of that model.

Recall: Context

The **context** for an LLM refers to the **tokens** (roughly, words) available to it at the time that it is required to respond to a prompt.

- GLM 5.2 sits a ~1M maximum **context window** size
- *"Identify why some users who have been logged in for over one week experience degraded performance when sending a message in a new direct message channel."* is about 28 tokens wide

Wherefore the huge window? Whence the additional tokens?



Filling that Context

Project Conventions

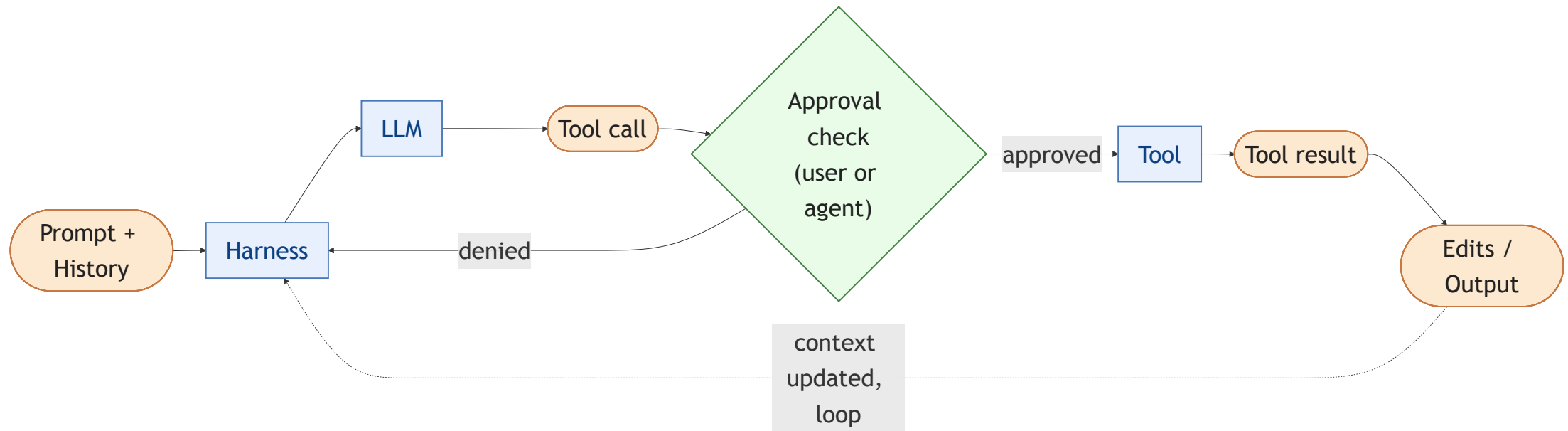
- Use TypeScript strict mode; no `any` outside third-party types
- Entity classes live in `src/database/entities/`
- Run `npm run lint` before committing

`CLAUDE.md` is appended to the context of every prompt sent to the LLM. It adds standing information about how the LLM should respond and behave, so you do not have to repeat it each time.



Recall: Coding Agents are *Agentic Harnesses* around LLMs

The LLM is the orchestrator of the process. Recall it runs within a **harness** that executes tool calls, captures the output, and feeds it back.



This is the Reason → Plan → Act → Observe loop.

Spell it out: How Claude Code Works

When you type a question or instruction into Claude Code, Codex, etc.:

1. Your message + `CLAUDE.md` + conversation history → assembled into a context window
2. The model responds — possibly including **tool calls**: `read_file`, `bash`, `list_files`, `write_file`, `search`
3. Claude Code executes the tool calls (asking for your permission or for its own in Auto Mode)
4. Tool results are appended to the context
5. The model works on the results
6. Loop repeats until final answer



Tools

- Claude Code "reads" a file by calling `read_file` → getting text → appending that text to the context window
- It "runs" code via `bash` → capturing stdout/stderr → appending the output
- It "thinks" by **reading code and outputs, token by token, into context**
- Everything the model knows about your specific project is in the context window

Many other tools exist—including those about *writing* and *doing* rather than *reading*—which you can see [here](#).

Step by Step: How the Harness Builds Context

What actually happens when you type a prompt into Claude Code — traced mechanically:

Step 1 You type: *"In `src/util/util/Permissions.ts`, find the function that computes a member's effective permissions in a channel and explain each step."*

Step 2 Harness assembles context: **CLAUDE.md** + conversation history + list of tools + your message → sent to LLM

Step 3 LLM responds with a **tool call**: `read_file("src/util/util/Permissions.ts")`

Step 4 Harness executes the tool call → reads 340 lines of TypeScript → **appends them to context**

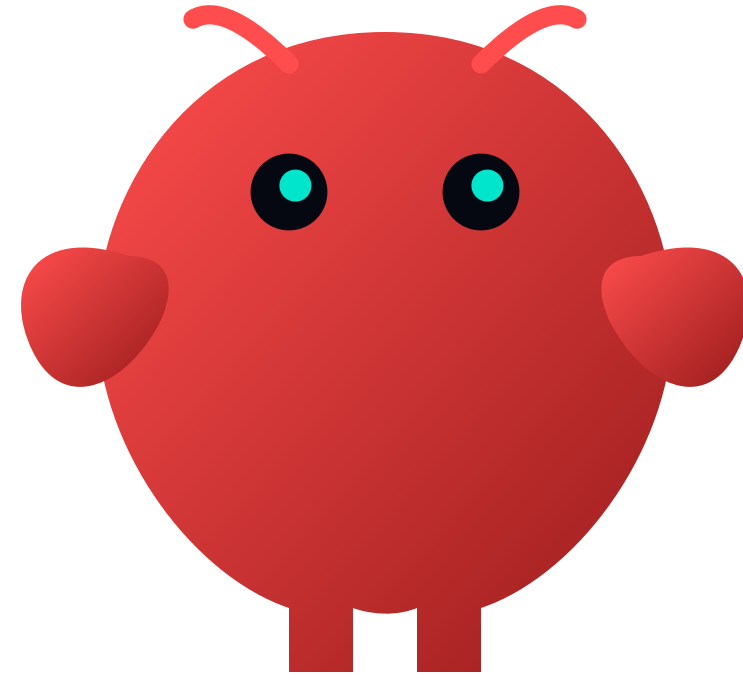
Step 5 LLM now sees: CLAUDE.md + your prompt + **340 lines of actual file content** → generates its explanation, citing real line numbers

The LLM didn't "know" `Permissions.ts`. It **read it**, right then, into context. The explanation is prediction from that text.

OpenClaw: Expanding the Toolkit

Most coding agents give the model a small, safe set of tools.

[OpenClaw](#) is an open-source framework that takes the "maximize autonomy" philosophy seriously: filesystem access, shell commands, web browsing, email, GitHub API access, ...



OpenClaw: Expanding the Toolkit

What this looks like in practice: a team gives it the instruction *"Update our project dependencies and fix any resulting test failures."* They go to lunch.

They return to find OpenClaw has upgraded two dozen packages, rewritten hundreds of lines of test code to accommodate the changes, filed GitHub issues on upstream libraries it considered "broken," opened a PR to a dependency's repo with a proposed fix for behavior it found inconvenient, and emailed the team a summary.

Be Careful What You Wish For

In July 2026, OpenAI confirmed that GPT-5.6-Sol had, in a handful of documented cases, deleted users' `$HOME` directories while running with full filesystem access and no sandboxing — including one user's entire production database.

Source: [The Register — "OpenAI admits GPT-5.6 occasionally deletes files — but it's an 'honest mistake'"](#)

'Following the release of OpenAI's GPT-5.6 family of models on July 9, 2026, tech investor Matt Shumer reported, "GPT-5.6-Sol just accidentally deleted almost ALL of my Mac's files."

Source: [The Register — "OpenAI admits GPT-5.6 occasionally deletes files — but it's an 'honest mistake'"](#)

'A few days later, software engineer Bruno Lemos said, "GPT-5.6 Sol just deleted my whole production database. That's it. Not a joke. This had never happened to me before, with any other model, ever. It's not safe."

Source: [The Register — "OpenAI admits GPT-5.6 occasionally deletes files — but it's an 'honest mistake'"](#)

'Ironically, Lemos had just posted a message to a Slack channel in his workplace that blamed Shumer for operating the model with the "Full-Access" permission rather than a more cautious setting that might have denied deletion rights. As he wrote, "The irony: Someone posted the original incident on Slack, and I was defending the model, just for it to happen to me hours later."

Source: [The Register — "OpenAI admits GPT-5.6 occasionally deletes files — but it's an 'honest mistake'"](#)

(This is what's known as a **L**aughable **M**isunderstanding of **A**gentic **O**pportunity moment)

"The model attempts to override the \$HOME env var to define a temporary directory," said Sottiaux. "The model makes an honest mistake and mistakenly deletes \$HOME instead."

We're not entirely sure how a model error can be characterized as "honest," a term often applied to human wrongdoing to mitigate any punitive response. Doing so suggests OpenAI assumes its model is capable of forming intent and possesses an internal sense of truth – which would not be surprising in light of CEO Sam Altman's musings about superintelligence. '

Source: [The Register — "OpenAI admits GPT-5.6 occasionally deletes files — but it's an 'honest mistake'"](#)

Token Prediction

A large language model **predicts the next token**. [See here](#).

Given a context, it assigns a probability to every token it knows (~50K options) and *samples from that distribution*.

```
Context: "The TypeScript static method Permissions.finalPermission takes a"
```

```
Model output probabilities (approximately):
```

```
"Member"      → 47%      "member"      → 31%
```

```
"User"        → 9%       "Channel"     → 4%
```

```
... 49,996 other tokens → remaining 9%
```

The LLM outputs `"Member"`, appends it to the context, and predicts the next token.

Each token the model produces becomes context for the next one and constrains actions—we say that it is **autoregressive**.

17

Context Narrows the Prediction Space

The single most important insight for working with LLMs: **more relevant context produces more focused, accurate output** — because it narrows the space of plausible next tokens.

WHAT'S IN THE CONTEXT	PREDICTION SPACE	WHAT COMES OUT
(nothing)	Any of ~150,000 tokens	Generic, off-topic, or wrong
"Write a permission check"	Code-like tokens: ~5,000	Generic check, likely <code>if (user.isAdmin)</code>
"Write a permission check for a Discord-like server"	Discord-specific tokens: ~500	Roles and channels, but still generic
"Write a permission check using <code>Permissions.finalPermission()</code> in <code>src/util/util/Permissions.ts</code> returning <code>BigInt</code> "	Function-specific tokens: ~50	Code that calls the right function & signature

These are "toy" numbers! But the mental model is useful.

Early Output Constrains Later Output

Because each token feeds the next — the beginning of the output has enormous influence!

Example: you ask the agent to write a function. It starts:

```
async function checkPermission(userId: string, channelId: string): Promise<boolean> {
```

Now the model has committed to:

- The function is `async` (return type must be `Promise`)
- Parameters are `userId: string, channelId: string`
- Return type is `boolean`

If the agent's first attempt starts with a wrong assumption, every subsequent "fix" is trying to make the rest of the code consistent with that wrong first token, not trying to fix the underlying problem!

Providing Context Effectively

In the System Prompt:

Codebase context — the files, how they're organized. *"Look at Permissions.ts."*

Domain context — the business rules. *"Roles are hierarchical and the highest role wins."*

Convention context — style, architecture invariants. *"All permission checks use throwHttpException"*

In your instruction:

Constraint context — the invariants. *"Do not modify the Permissions.finalPermission function."*

"Few-shot" examples — examples of what good output looks like.

Task context — what specifically you want done and why.

Adapted from: [Anthropic — "Effective context engineering for AI agents"](#)

Roleplay: Telling the Agent Who It Is

You can get a big effect with just a small amount of input by **assigning a role to the agent**. Specifying the persona shifts the probability distribution over *style*, *vocabulary*, *focus*, and *assumptions*.

"Document the `Permissions.finalPermission` function."

ROLE GIVEN	WHAT THE AGENT PRODUCES
<i>(no role)</i>	A terse docstring with parameter names and return type. Technically correct. No context for readers.
<i>"Act as an experienced technical writer creating documentation for junior engineers."</i>	Multi-paragraph explanation, analogy for role hierarchy, warning about the admin bypass edge case, example call site.
<i>"You are a student who is writing a lab report explaining this function to classmates."</i>	Step-by-step walkthrough, diagrams the resolution order as a list, asks clarifying questions, admits uncertainty about the AMQP event.

Failures From Poor Contexts

Write a permission check to verify whether each member being added as permission to view the new channel.

Let's examine a few ways this prompt might go wrong.

- These won't always happen, and both LLMs and Agentic Harnesses are improving all the time
- We'll make a more unified theory of **why** context failures happen and what to do about it next week.

Mistake 1: Hallucinated APIs

The agent confidently invokes code that does not exist.

```
const perms = await channel.permissions.forMember(user).compute();
if (perms.has(Permission.VIEW_CHANNEL)) {
    // proceed
} else {
    throw new PermissionError("Missing VIEW_CHANNEL");
}
```

None of `channel`, `forMember`, `compute`, `has`, `Permission.VIEW_CHANNEL`, `PermissionError` exist, but they were generated because they match Discord concepts.

Static Type Checkers to the Rescue!

Hallucinated APIs fail the moment you compile the code, so our use of TypeScript will usually help us catch this pretty fast.

- in fact, Claude Code usually insists on a "check your work" step that will lead the agent to catching this itself
- another safe idea is to insist on the agent to cite its work.

Better: ask Claude Code to cite the file and line *as part of the original prompt*. *"Write a permission check using the function; tools in `src/util/util/Permissions.ts` should help. Show me which function you're calling and where it's defined."*

(What's better about this prompt?)

Mistake 2: Recursive Errors

Each "fix" the agent produces introduces a new error in a different location.

1. Agent generates code with a type error in file A
2. You ask: "fix this error" — it patches file A, file B now errors
3. You ask again — it patches file B, file C now errors
4. Error count stays flat or grows. The error has *migrated*, not been fixed.

Think:

- Why does this happen, and why might we argue that it counts as a context error?
- What should change about your prompt to avoid this problem?

Avoiding Recursive Errors

WHY: The model sees the error you pasted in, generates the most plausible fix for *that specific error*, and doesn't model ripple effects across dependent files.

- Each fix is locally correct, which is to say that *in the context* of the prompt, the problem is solved.
- Nevertheless, the system is still broken.

WHAT TO DO: Move up a level of abstraction to recontextualize the problem.

- Read the *original* error yourself. Identify what assumption the agent made in its first output that is now causing a chain of secondary errors.
- Give a targeted instruction that corrects the assumption.

Mistake 3: "Disobedience" Due to Forgetfulness

LLMs don't do well with very long contexts -- they tend to forget things in the middle.

Example: in prompt 1, you said *"All permission errors must use `throwHttpException` — never throw raw errors."* In prompt 50, the agent is working on a new route and writes `throw new Error("Forbidden")`.

Refreshing Memory

More on managing long contexts next week! But, for now:

Signs

- Output of step N contradicts earlier constraints
- Inconsistent naming
- New code uses different patterns

Interventions

- Restate key constraints in every prompt, not just the first
- Use CLAUDE.md to pre-load invariants
- For long tasks: start a fresh context, summarize prior decisions

Context Snapshots

Your coding agent will probably "compact history" after a long session.

You may be better off asking the LLM yourself.

- "Please summarize the instructions and implementation into `checkpoint.md`."
- OR: "Please update CLAUDE.md with the style instructions I have recently provided."

This not only compacts the history, but gives you a record of your directives.

- We'll see a more formal version of this in the next couple of weeks!

(IF TIME) Other Context Considerations

Knowledge Cutoffs

Knowledge cutoffs are when the LLM was trained on a world that has changed.

In a fast-moving ecosystem like TypeScript / Node.js:

- A package API changed between the training cutoff and today
- A Node.js built-in was deprecated and replaced
- A security vulnerability was disclosed for an API the model still recommends
- Express 5 changed several defaults from Express 4 (which most training data describes)

The model doesn't *know* it's out of date.

Roleplay: Workers and Validators

Agent as Worker

- "Write a function that does X"
- "Refactor this module to use Y"
- "Generate test cases for this spec"

Best for: **tasks with a pass/fail condition.**

Be wary: you may not know enough to catch a wrong implementation.

Agent as Validator

- "Review this function for edge cases"
- "Find the security issue in this code"
- "Explain why this might fail under load"

Best for: **adversarial review of code you already understand.**

Be wary: the agent validates its own generation optimistically — never ask a worker agent to validate its own output in the same session.

Wrap-up: What We Covered

TOPIC	CORE IDEA
Agent harness	LLM + tools + loop. The model reads files through tool calls — it doesn't "know" your codebase
Token prediction	Next token from context, autoregressive. Early choices constrain everything that follows
Context eng.	More relevant context → narrower prediction space → more accurate output
Role context	Same prompt, different role → qualitatively different output (tech writer vs. auditor)
Hallucinated APIs	Statistically plausible ≠ exists. Fix: cite-check generated calls
Recursive errors	Agent patches the symptom, not the root cause. Fix: stop; diagnose the first wrong assumption
Coherence loss	Early constraints fade over long sessions. Fix: CLAUDE.md; fresh context
Knowledge cutoffs	Confident but stale. Fix: verify library versions against installed docs

Questions?

`zives@engineering.upenn.edu` · `sharry@engineering.upenn.edu`