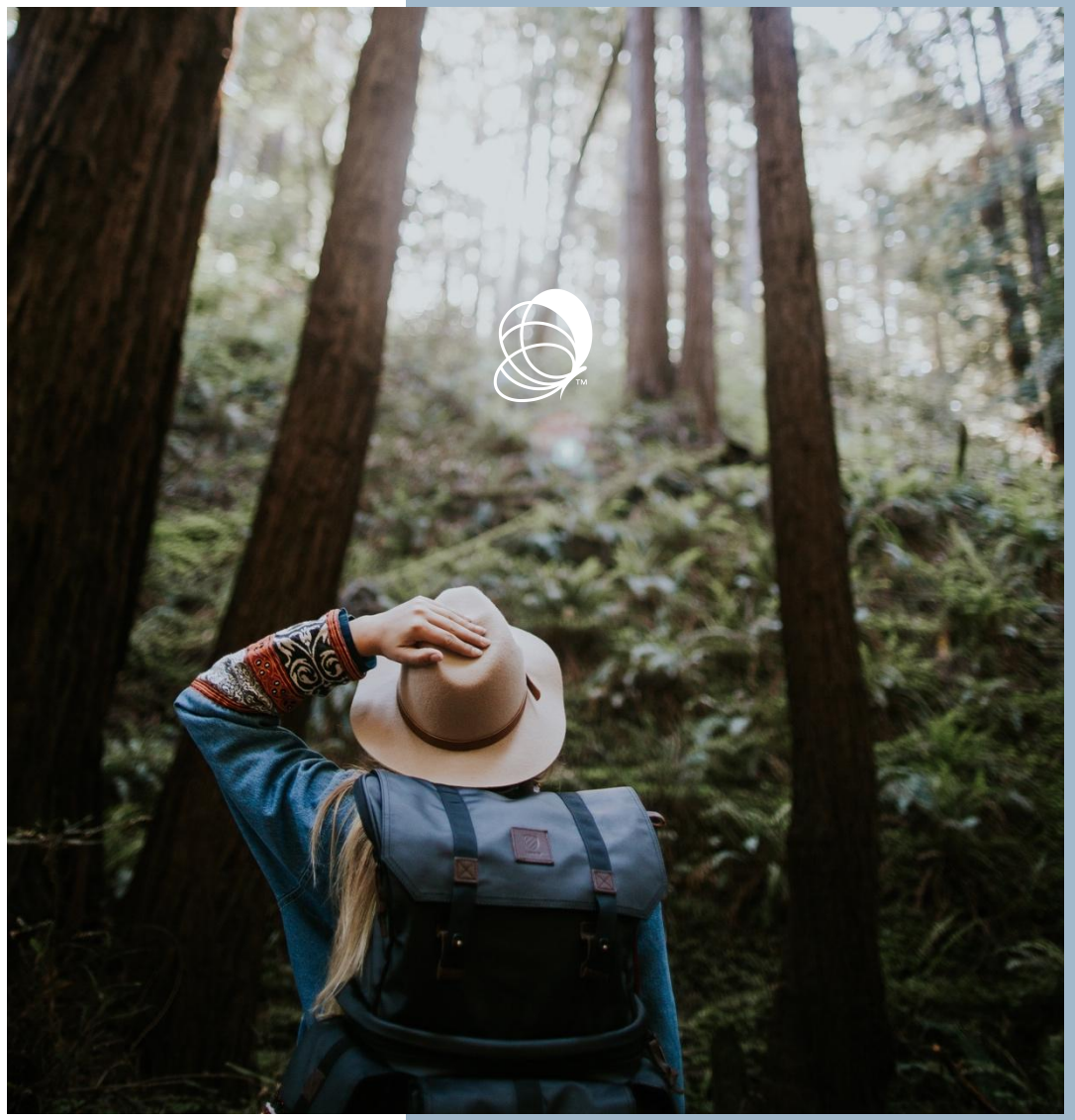
Agentic Vibe Coding from the Terminal

Tim Oates

April 2, 2026

Outline

- The big picture
- How to talk to coding agents
- Putting it to practice
  - A simple game
  - A tool for visualizing ML results
- Getting the most out of Claude Code



Why and What

In 60 minutes you'll see how AI coding agents turn "I wish we had a tool for that" into a working prototype

A goal-driven assistant that can plan, write code, run it, and fix it, all under your supervision

You're the CEO

"Our sales tracking system is a mess! I keep having to check for problems and nag the sales team to fix them. I wish we could automate checking and push issues to them."

Call the IT department

You're the CEO

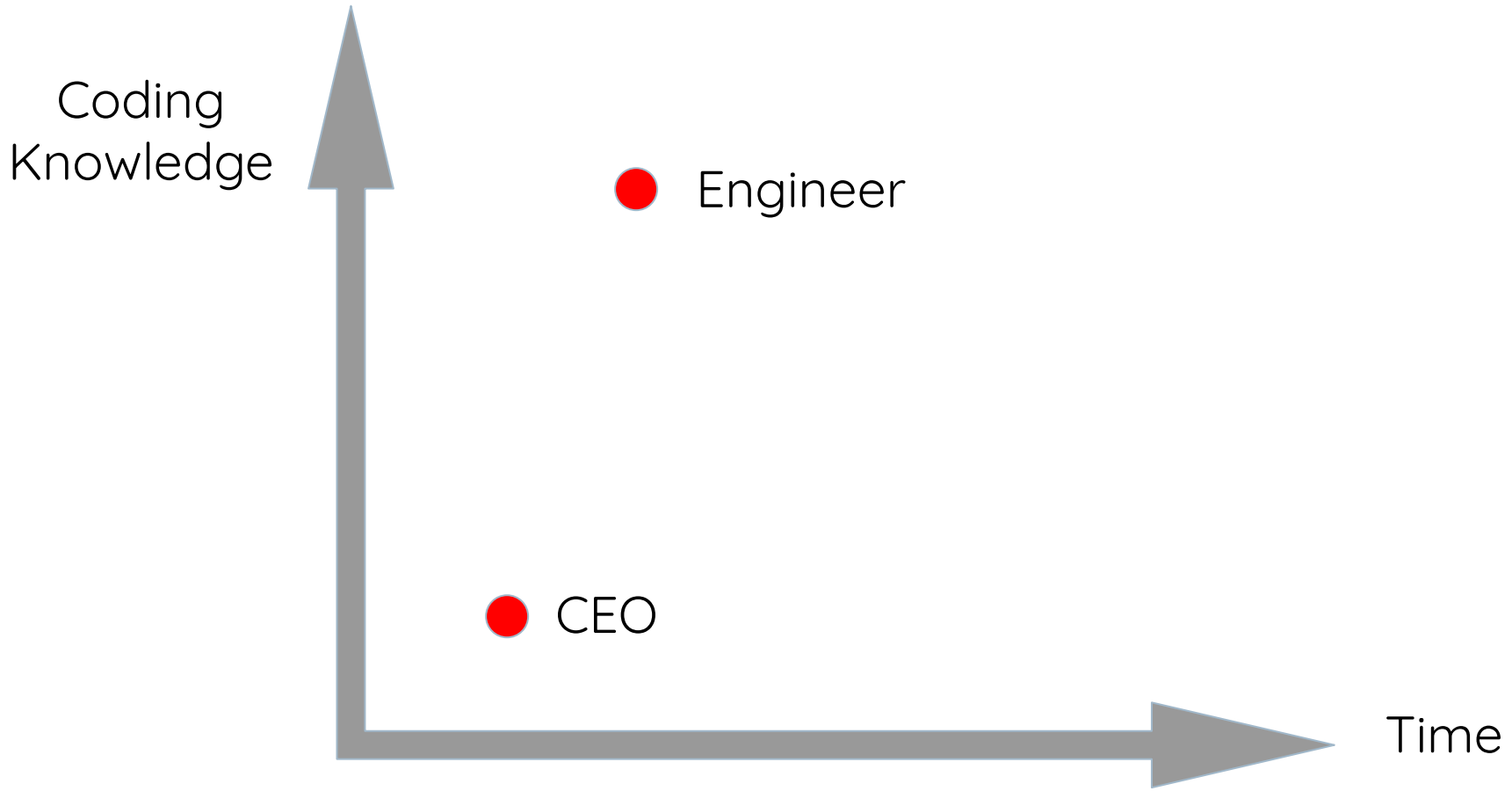
"Our sales tracking system is a mess! I keep having to check for problems and nag the sales team to fix them. I wish we could automate checking and push issues to them."

Talk to a coding assistant

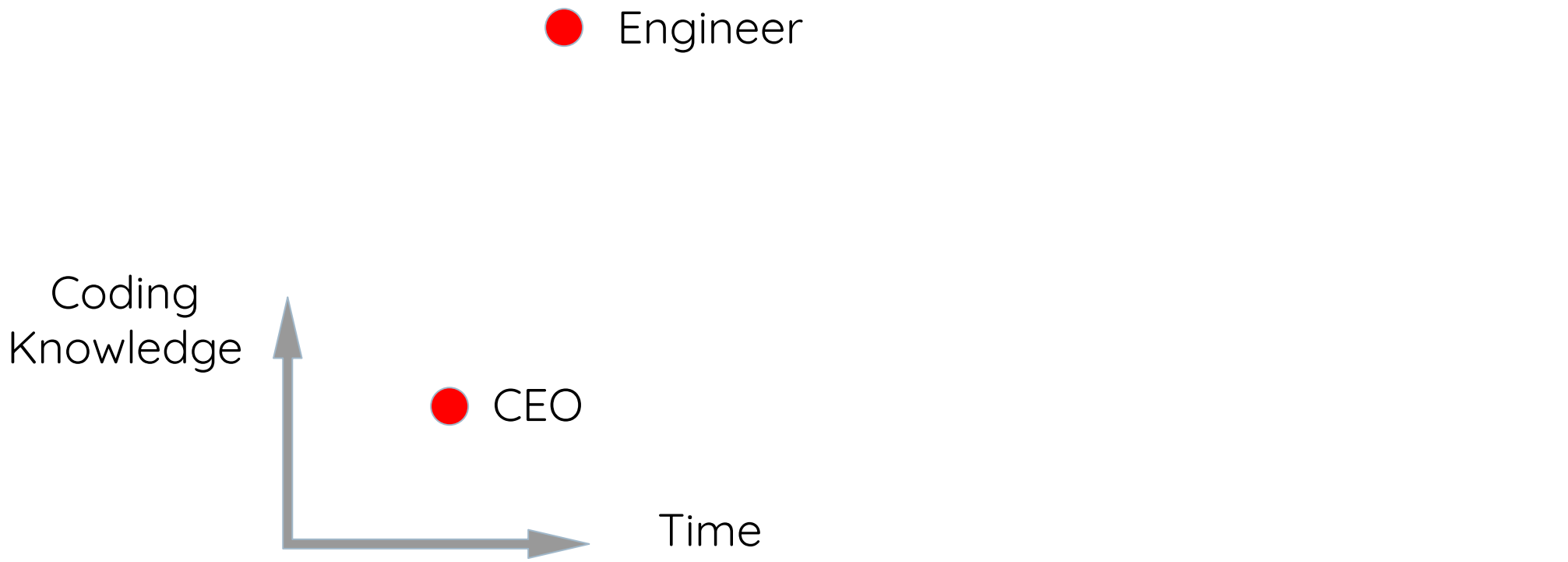
"Our sales database has lots of missing data, duplicates, and bad dates. I'd like to create a system that checks it every night for common errors and sends emails to the responsible salesperson with a list of the things that need to be fixed."

OK, I've got a few questions …

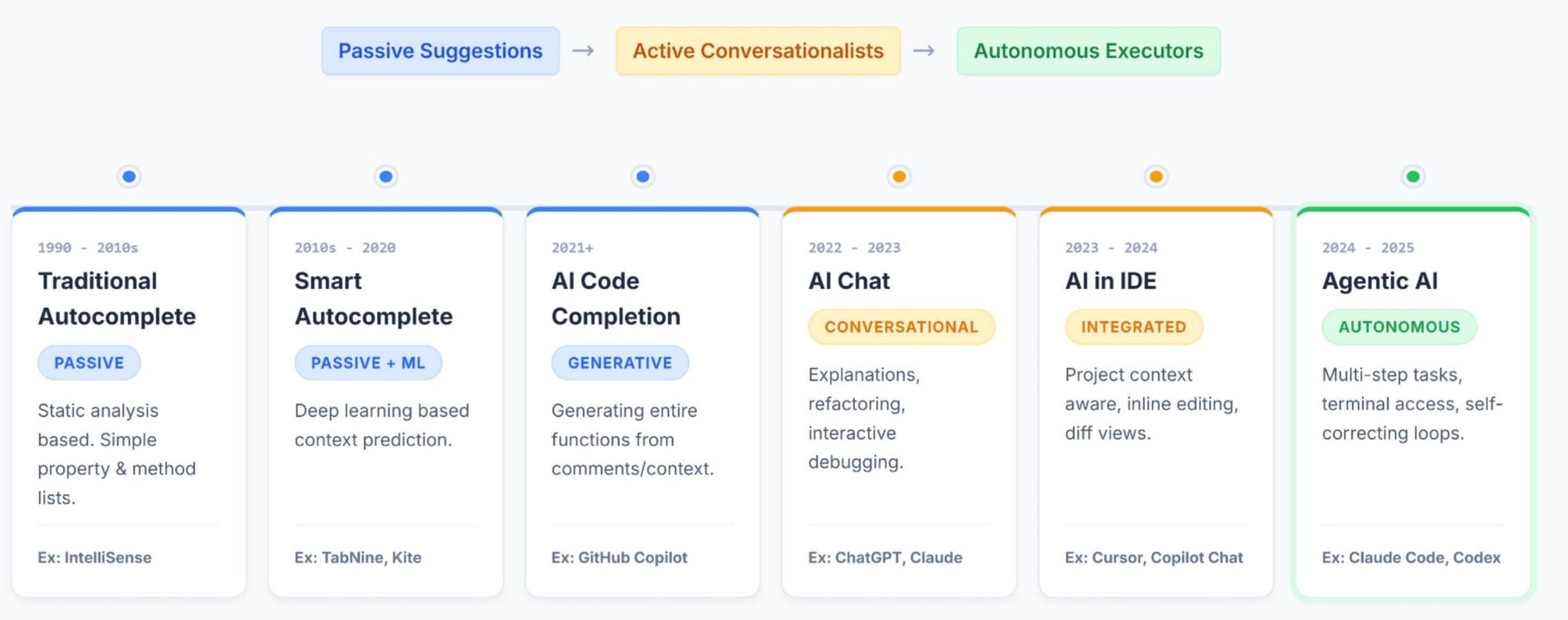
Things Needed to Write Software



Things Needed to Write Software with Coding Assistants



Evolution of Coding Assistants



Your Mindset

The assistant is a **very**

- talented
- knowledgeable
- eager

graduate from MS/PhD **many** different computer science programs

Your Mindset

Stages of development

- describe what you want
- resolve ambiguity
- create spec
- code to spec
- run code and iterate to
  - fix bugs
  - polish
  - deploy

Your Mindset

Three principles

- Find your level of trust
- **Don’t turn off your brain**
- Remeber to say “Can you do that?”

There Are Lots of Tools

Claude in the News

TECHNOLOGY | ARTIFICIAL INTELLIGENCE Follow

Anthropic Races to Contain Leak of Code Behind Claude AI Agent

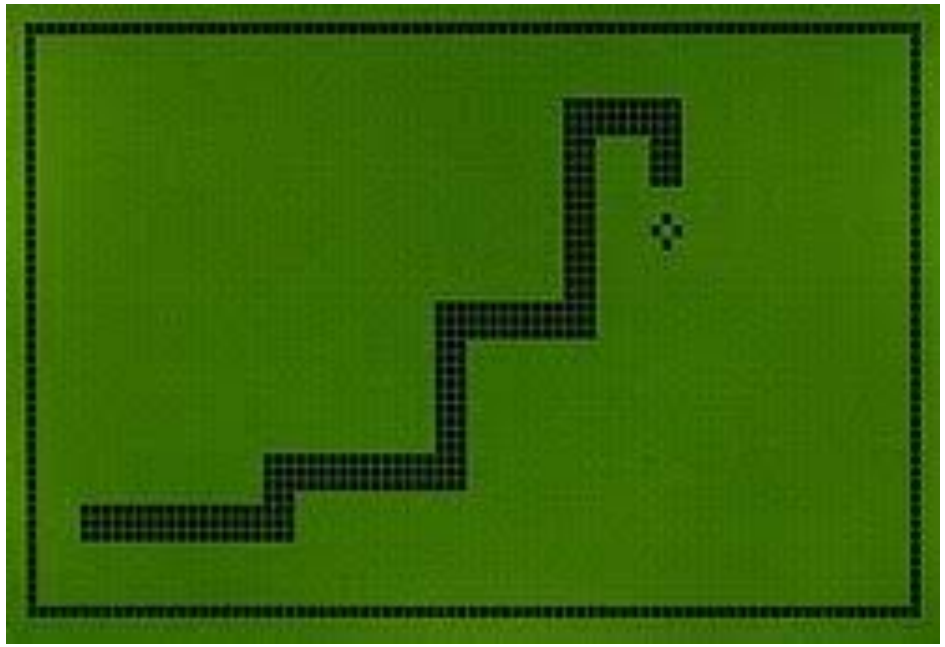
Developer issues copyright takedown request in bid to prevent competitors from cloning coding tool's features

ARTIFICIAL INTELLIGENCE

Critical Vulnerability in Claude Code Emerges Days After Source Leak

Within days of each other, Anthropic first leaked the source code to Claude Code, and then a critical vulnerability was found by Adversa AI.

Let's Get Started!

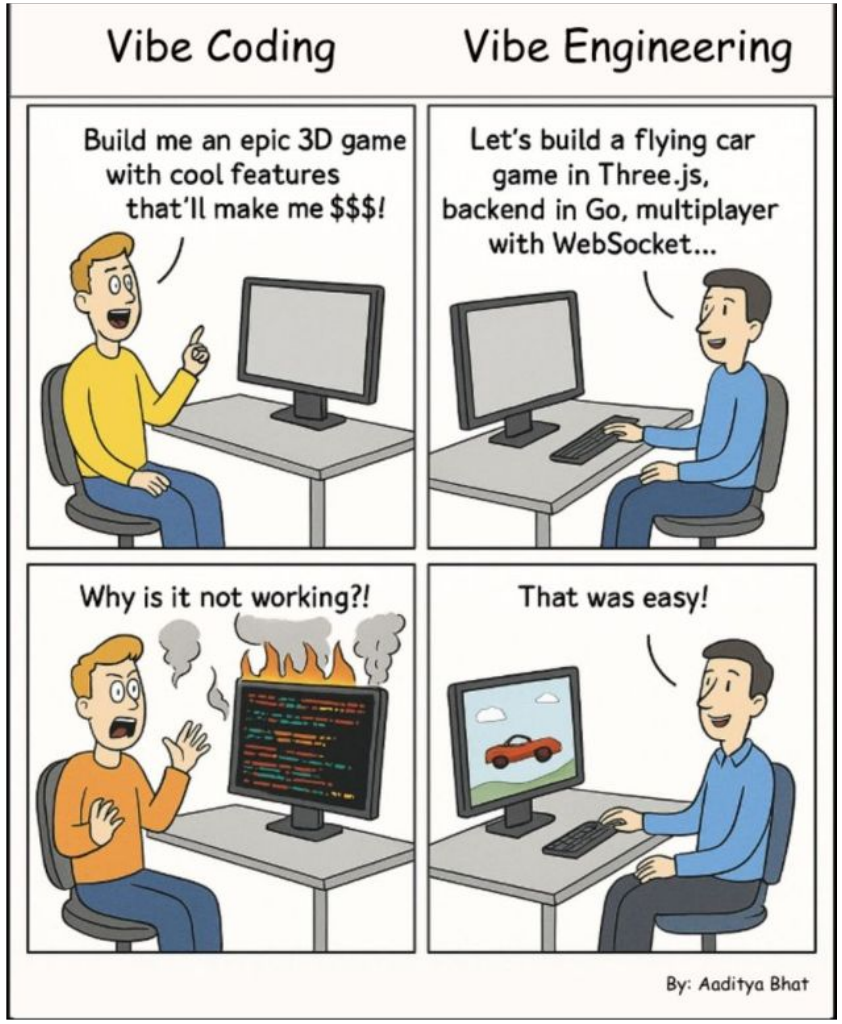


Stages of development

- **describe what you want**
- resolve ambiguity
- create spec
- code to spec
- **run code and iterate** to
  - fix bugs
  - polish
  - deploy

Live Demo

Vibe Coding vs. Engineering



Planning Mode

Stages of development

- describe what you want
- **resolve ambiguity**
- **create spec**
- **code to spec**
- run code and iterate to
  - fix bugs
  - polish
  - deploy

The Prompt

```
/plan Build a Streamlit app called ML Explorer.
The user picks a dataset (iris, wine, breast
cancer), picks a classifier (KNN, SVM, random
forest, logistic regression), and adjusts
hyperparameters with sliders. The app should show:
a 2D PCA projection with decision boundaries, a
confusion matrix heatmap, and
accuracy/precision/recall metrics. Everything
updates live when the user changes a setting.
```

Live Demo

CLAUDE.md: Suggestions for Claude

Injected into system prompt
Treated as natural language

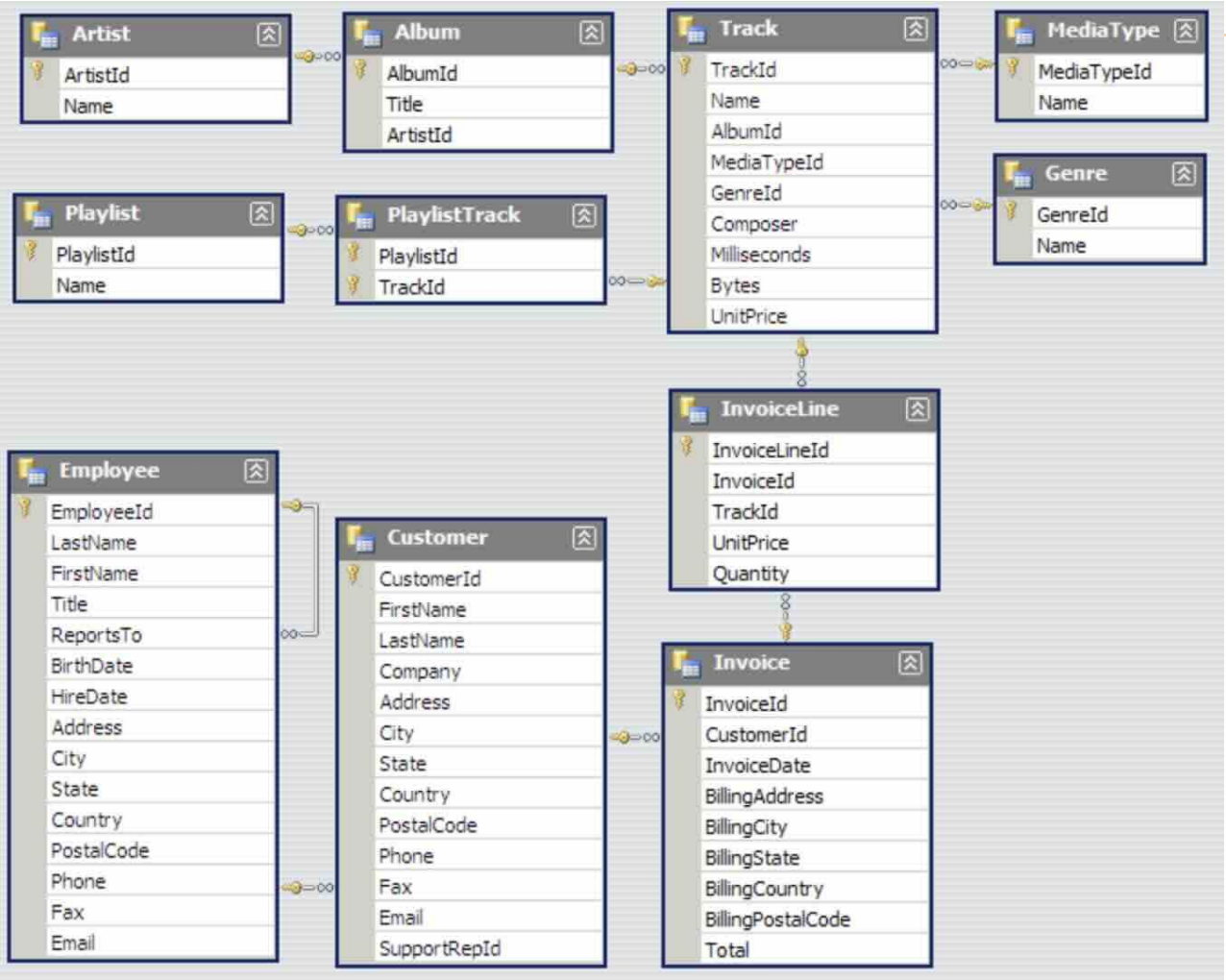
```
## Commands
- Run the app: `streamlit run app.py`
- Run tests: `pytest -x`
- Format a file: `black <filename>`
```

```
## Conventions
- Use type hints on function signatures
- Write docstrings in Google style
```

CLAUDE.md in the project root, in any subdirectory, in home directory

Demo With a Database

Chinook: a digital media store, including tables for artists, albums, media tracks, invoices and customers.



CLAUDE.md

```
## Conventions

- Always include type hints on function signatures and return
types
- Write docstrings in Google style for every function
- Use pathlib instead of os.path
- Always include error handling for database operations
- When writing SQL, use parameterized queries, never string
formatting
- Include a `if __name__ == "__main__"` block with example usage
```

Live Demo

Model Control Protocol: Turning AI Into Action

Tool Use

What is a Tool?

A function exposed by an MCP server that the LLM can call

Each tool has:

- A **name** — so the LLM can invoke it
- A **description** — so the LLM knows when to use it
- **Parameters** — the inputs it expects

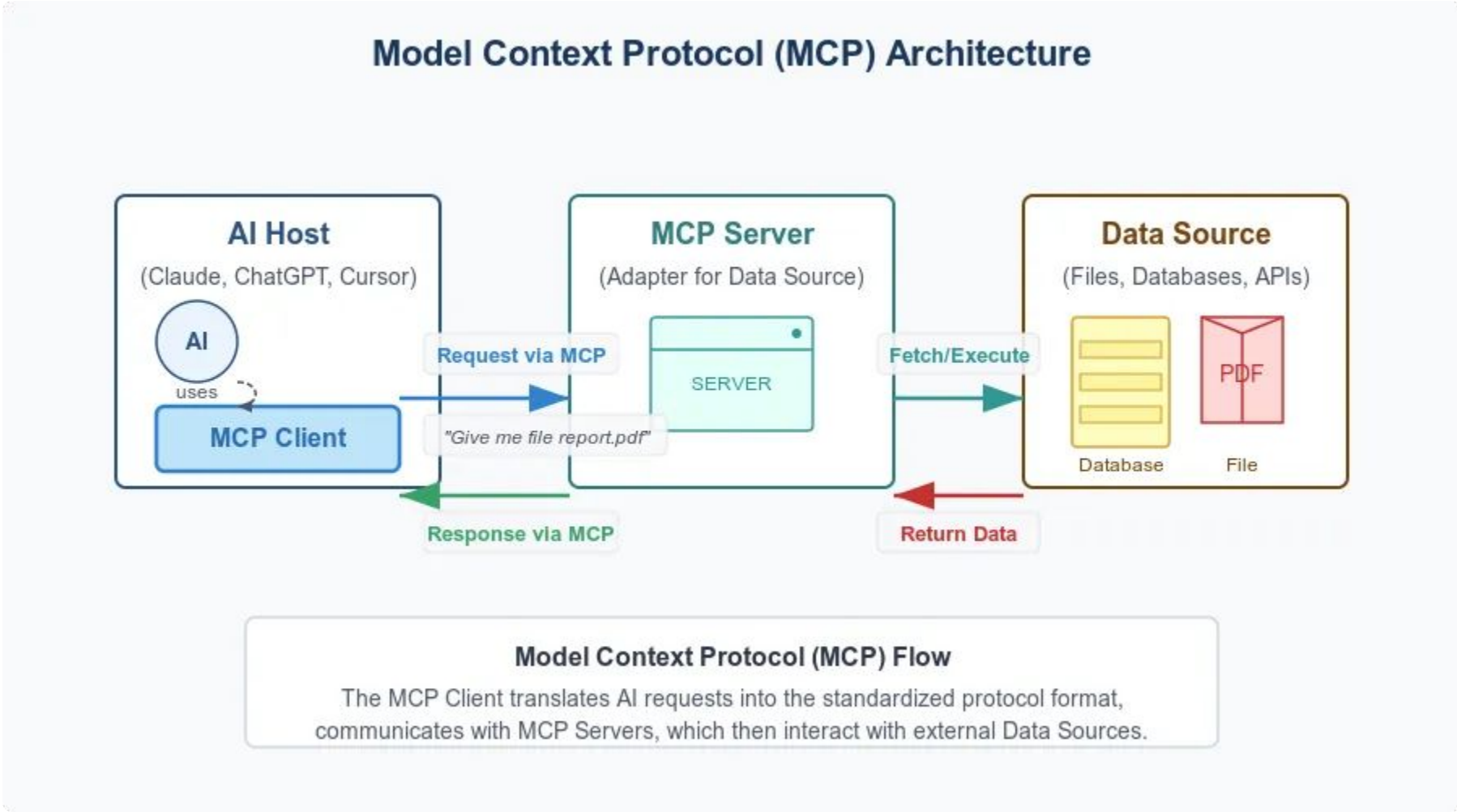
How it works:

- LLM decides to call a tool (e.g. read_query)
- Claude Code sends a JSON-RPC request to the MCP server
- MCP server executes the operation
- Result is returned to the LLM

Why it matters:

- Extends the LLM beyond text generation into taking actions
- Structured, typed operations instead of raw shell commands
- MCP server manages the connection and execution

MCP Architecture



Live Demo

Hooks: Rules Claude Cannot Ignore

"If this happens, run that shell command"

Event: what triggers the rule

- PreToolUse
- PostToolUse
- Stop - Claude finishes response

Matcher: which tool does it apply to

- Regex on tool name
- "Edit|Write"
- "Bash"

Hook command: what command to run

- Any shell command
- Gets JSON via stdin with details
- Exit = 0 means OK
- Exit = 2 means block

Hooks: Rules Claude Cannot Ignore

```
{
  "hooks": {
    "PostToolUse": [
      {
        "matcher": "Edit|Write"
        "type": "command",
        "if": "Edit(*.py)|Write(*.py)",
        "command": "jq -r '.tool_input.file_path' | xargs black"
      }]}]}}
```

Claude knows the syntax
You need the idea

~/.claude/settings.json
- All your projects

./.claude/settings.json
- This project

Live Demo

Test Driven Development

The sweet spot: LLMs are good at TDD because the workflow is structured

- Write tests first, then implement until they pass.

"Spec first" prompting

- Write your tests (or describe expected behavior) *before* asking the LLM to implement
- The tests act as an unambiguous spec; less room for misinterpretation than prose

Why it works so well with LLMs

- Tests are a *verifiable* contract: you don't have to trust the output, you run it
- Catches hallucinated logic immediately
- Gives the LLM a tight feedback loop: fail → fix → pass

THANKS
FOR
LISTENING