Building Claude Code from Scratch: A Simple Journey into AI Agents

6 min read · 2 days ago

JBY Follow

What I learned by building my own AI coding assistant

When I started this project, I had a simple goal: understand how tools like Claude Code, Cursor, and Gemini CLI actually work. After building a working AI coding agent in ~300 lines of Python, I learned something surprising: **you don't need a framework to build an AI agent.**

In this article, I'll show you exactly how to build your own AI coding agent from scratch using just the Claude API, Python, and some fundamental patterns. By the end, you'll have a working agent that can:

- Read and write files autonomously
- Search the web for information
- Execute shell commands
- Remember conversation context
- Make decisions about which tools to use

But more importantly, you'll understand when to use frameworks like LangChain and when to just write your own code.

The Big Question: Framework or From Scratch?

Before I started building, I spent hours researching whether I should use LangChain, LangGraph, or CrewAI. The documentation was intimidating. The

abstractions were complex. I felt like I needed a PhD to understand multi-agent orchestration.

Then I found this gem from a 2025 article:

> *"Companies often adopt 'unearned complexity' by deciding on LangChain or multi-agentic solutions without experimenting enough to understand if they actually need that complexity."*

So I decided to take a different approach: build the simplest thing that could possibly work, then add complexity only when needed.

Spoiler: I never needed the framework.

What We're Building

Our AI agent will have:

1. **5 Tools** — read files, write files, edit files, execute shell commands, search the web

2. **ReAct Loop** — The industry-standard pattern for autonomous agents

3. **Conversation Memory** — Maintains context across multiple tasks

4. **Colored CLI** — Because good UX matters

5. **Human-in-the-Loop**— Safety confirmations for dangerous operations

The entire project is ~300 lines across 4 files. No frameworks. No abstraction layers. Just clean, understandable code.

Understanding the ReAct Loop

The core of any AI agent is the ReAct (Reason + Act) pattern. It's surprisingly simple:

1. User gives a task
2. Loop:
   a. Agent THINKS about what to do
   b. Agent chooses a TOOL to use
   c. Tool executes and returns RESULT
   d. Agent OBSERVES the result
   e. Repeat until task is complete

3. Agent provides final answer

This is the same pattern used by Claude Code, ChatGPT Code Interpreter, and every other coding agent.

Building Block 1: The Tools System

Every AI agent needs tools to interact with the world. Let's start with the simplest possible implementation.

Tool Schema (for Claude API)

Claude needs to know what tools are available and how to use them. We define this using JSON schemas:

```
# tools.py
TOOL_SCHEMAS = [
    {
        "name": "read_file",
        "description": "Read the contents of a file.",
        "input_schema": {
            "type": "object",
            "properties": {
                "path": {
                    "type": "string",
                    "description": "The file path to read"
                }
            },
            "required": ["path"]
        }
    },
    # ... more tools
]
```

Tool Implementation

Each tool is just a Python function that returns structured data:

```
def read_file(path: str) -> Dict[str, Any]:
    """Read a file and return its contents."""
    try:
        with open(path, 'r') as f:
            content = f.read()
        return {
            "success": True,
            "content": content
```

```
            }
        except Exception as e:
            return {
                "success": False,
                "error": str(e)
            }
```

That's it. No abstractions. No inheritance. Just a function that does one thing.

I built 5 tools this way:

- **read_file** — Read file contents
- **write_file** — Create or overwrite files
- **edit_file** — Find and replace text
- **shell_command** — Execute shell commands
- **web_search** — Search the web using DuckDuckGo

Building Block 2: The Agent Core

Now for the heart of the system: the ReAct loop.

```
class CodingAgent:
    def __init__(self, api_key: str):
        self.client = Anthropic(api_key=api_key)
        self.conversation_history = []

    def run(self, user_prompt: str, max_iterations: int = 15):
        # Add user's prompt to conversation
        self.conversation_history.append({
            "role": "user",
            "content": user_prompt
        })

        iteration = 0
        while iteration < max_iterations:
            # Call Claude with available tools
            response = self.client.messages.create(
                model="claude-sonnet-4-5",
                tools=TOOL_SCHEMAS,
                messages=self.conversation_history
            )

            # Check if Claude wants to use a tool
```

```
            if response.stop_reason == "tool_use":
                # Execute the tool
                tool_result = execute_tool(...)

                # Add result to conversation
                self.conversation_history.append({
                    "role": "user",
                    "content": tool_result
                })

            elif response.stop_reason == "end_turn":
                # Task complete!
                return response.content

            iteration += 1
```

This is the entire ReAct loop. Claude decides when to use tools, which tools to use, and when it's done. Our job is just to:

1. Call the API

2. Execute tools when requested

3. Feed results back

4. Repeat until done

Building Block 3: The CLI Interface

We need a way for users to interact with the agent. A simple loop with conversation memory:

```
# main.py
def main():
    agent = CodingAgent(api_key=os.getenv("ANTHROPIC_API_KEY"))

    while True:
        user_prompt = input("> ")

        if user_prompt == "exit":
            break

        # Run the agent
        result = agent.run(user_prompt)
        print(result)
```

That's the basic version. I added:

- Colored output (using **colorama**)

- **/reset** command to clear history

- **/history** command to see conversation stats

- Safety confirmations for dangerous operations

The Web Search Addition

Here's where it gets interesting. Adding a new capability is trivial.

To add web search, I:

1. Added **ddgs** package to requirements

2. Created a **web_search** function

3. Added the schema to **TOOL_SCHEMAS**

That's it. The agent immediately knew how to use it:

```
def web_search(query: str, max_results: int = 5):
    """Search the web using DuckDuckGo."""
    try:
        ddgs = DDGS()
        results = ddgs.text(query, max_results=max_results)
        return {
            "success": True,
            "results": results
        }
    except Exception as e:
        return {
            "success": False,
            "error": str(e)
        }
```

No framework needed. No complex integrations. Just a function.

The Moment of Truth: Is This Actually an Agent?

After building this, I had a crisis: "Did I just build an LLM wrapper?"

So I researched the 2025 definitions of AI agents. Here's what I found:

What Makes Something an AI Agent (According to IBM, 2025):

Autonomous decision-making — Claude decides which tools to use

Tool use — Interacts with the environment

ReAct loop — Think → Act → Observe

Goal-driven— Works toward completing tasks

Perceives environment — Reads files, searches web

Takes actions — Actually modifies files and runs commands

My code checked every box.

The Framework Question: When Do You Actually Need One?

After building this, I researched when developers typically switch to frameworks. Here's what I found:

You DON'T need a framework if:

- You have < 10 tools

- You're building one agent (not a team)

- Simple conversation memory works fine

- You don't need RAG (vector database search)

You MIGHT need a framework when:

- You need RAG with semantic search

- You need 3+ agents collaborating

- You need 20+ external integrations

- You need complex state machines

- You're going to production with enterprise features

Key Lessons Learned

1. Start Simple, Add Complexity When It Hurts

I almost started with LangChain because “that’s what professionals use.” But by building from scratch, I:

- Understood exactly how agents work

- Can debug every line of code

- Have zero framework lock-in

- Learned the patterns that frameworks abstract away

2. The ReAct Loop Is All You Need

Every coding agent — from Claude Code to GitHub Copilot — uses this same loop. Understanding this is more valuable than knowing any specific framework.

3. Tool Calling Is the Superpower

The magic isn’t in the LLM’s intelligence. It’s in giving it:

- The ability to take actions (tools)

- A feedback loop (observations)

- The autonomy to iterate

4. “Agent” Is Simpler Than You Think

You don’t need:

- Multi-agent orchestration

- Vector databases

- Complex memory systems

- Graph-based workflows

You just need:

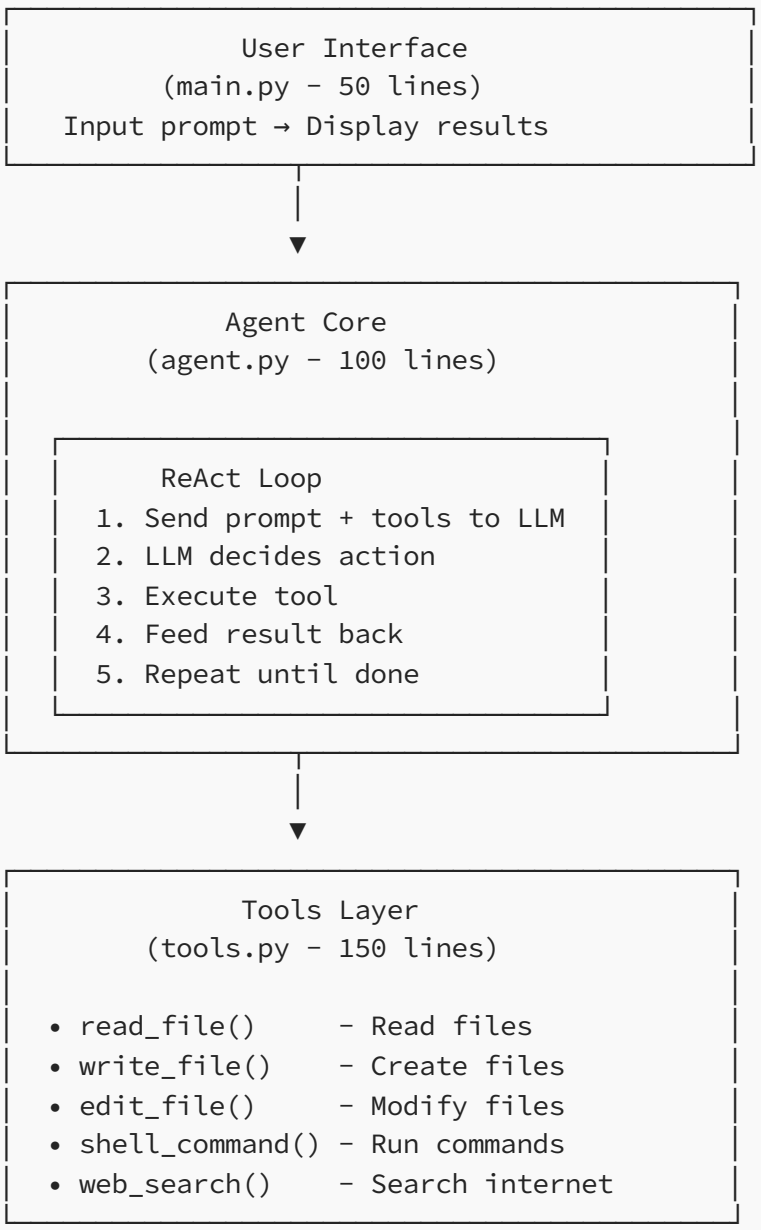
- An LLM that can call tools

- A loop that feeds results back

- Tools that do real things

The Full Architecture

Here's the complete picture of what we built:



Conclusion: You Don't Need Permission

The AI agent landscape is full of intimidating terms: multi-agent orchestration, vector embeddings, semantic search, graph-based workflows, RAG pipelines.

But at the core, an AI agent is just:

1. An LLM
2. Some tools
3. A loop

You just built one in 300 lines.

You don't need:

- A PhD in machine learning
- A complex framework
- Months of learning
- Permission from the "AI engineering" community

You just need:

- An API key
- Python basics
- The ReAct pattern
- The willingness to start simple

The frameworks exist to solve problems you don't have yet. Build first. Abstract later. Learn continuously.

And when someone asks "Why not just use LangChain?", you can say:

"Because I wanted to understand how it actually works."

Resources

Code:

- Full implementation: https://github.com/yashv6655/Simple-Claude-Code

Source: https://devblogs.microsoft.com/ise/earning-agentic-complexity/

Ai Agent | Python | Anthropic Claude

Follow

Written by JBY

0 followers · 1 following

Just another tech nerd

Responses (1)

Bgerby

What are your thoughts?

Yash Kathoke
6 hours ago

nice one bro

Reply

More from JBY

JBY

The great AI coding showdown: Claude Code vs Gemini CLI

The AI coding assistant market just got a major shakeup. Google's Gemini CLI launched on June 25, 2025, with an impossibly generous free...

Jun 26

JBY

The Great Compression: Why AI Models Are Getting SmallerAnd Smarter

In the early days of large language models, bigger meant better. GPT-3, released in 2020 with 175 billion parameters, was hailed as a...

May 22

JBY

Apple FastVLM: On-Device Multimodal AI Without the Cloud

In May 2025, Apple quietly open-sourced a project that could redefine the future of multimodal artificial intelligence.

May 12

See all from JBY

Recommended from Medium

Ahmed Shika Shaker

The AI Development Revolution: Building Intelligent Agent Swarms with Claude Code

The landscape of software development is undergoing a seismic shift. We're not just talking about AI-assisted coding anymore—we're...

Oct 13

In The Context Layer by Jannis

Claude's New "Skills" Show How Anthropic Is Layering Intelligence on Top of MCP

3d ago 1

Riccardo Tartaglia

5 Essential MCP Servers Every Developer Should Know

I've been experimenting with Model Context Protocol servers for a few months now, and I have to say, they've changed the way I work.

Oct 11 2

Hash Block

7 n8n Playbooks That Kill Repetitive Engineering Work

Reusable, battle-tested workflows that automate PRs, alerts, releases, and on-call housekeeping—so your team ships faster with fewer...

4d ago

In AI Mind by Adham Khaled

I Spent $200 on Claude Last Month. Then I Found GLM-4.6

How Z.ai's new 355B parameter model delivers enterprise-grade coding at 1/7th the cost—and why embedded engineers like me are switching

6d ago 2

Luis M. Gallardo D.

GitHub Copilot CLI: Terminal-Native AI with Seamless GitHub Integration

I stumbled upon GitHub Copilot CLI while catching up on tech news, and the native GitHub integration immediately caught my attention. A...

Oct 1

See more recommendations