# Build a Local AI Coding Agent (No Cloud Needed)

Make your own coding assistant!

Louis-François Bouchard Follow 9 min read · 5 days ago

Most coding assistants send your code to external servers, even for simple tasks like reading files or running shell commands.

That's a problem if you care about privacy, work in secure environments, or just want full control.

In this post, we'll build **Local Cursor** — a terminal-based AI coding agent that runs entirely offline using open-source models.

> ***Note:*** *The term "local" is often misused. Here, it means everything — from model inference to file access — runs entirely on your machine. No API calls, no cloud dependencies.*

It will:

- Use Ollama to run a local LLM (we'll use `qwen3:32b`)
- Handle tool calls for reading/writing files, listing directories, and running shell commands
- Chain multiple tools to solve multi-step tasks
- Be easily extendable with things like web search or formatters

Local Cursor has three core components:

**1. CLI Interface**

Built using `click`, a lightweight Python library that makes it easy to define commands and options. This lets you chat with the agent directly in your terminal.

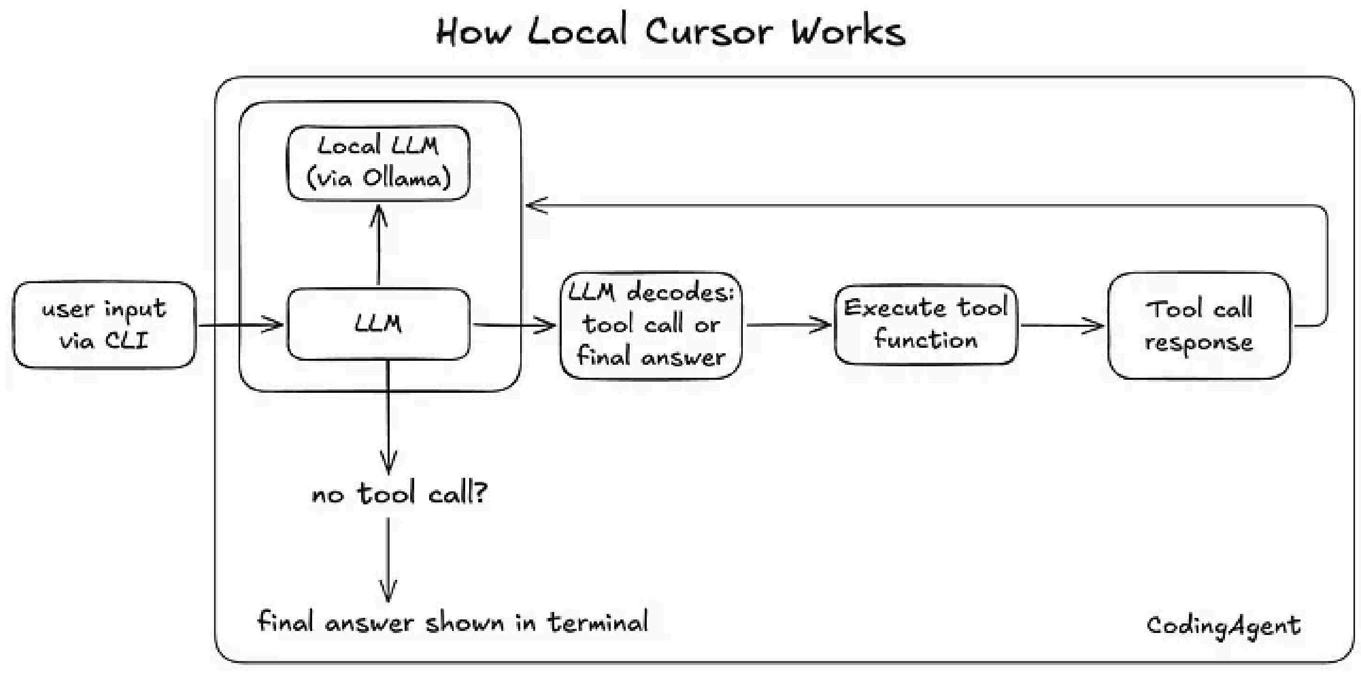
**2. Ollama Runtime**

We use `qwen3:32b`, a fast, open-source reasoning model that runs fully offline via Ollama.

**3. Coding Agent**

This is the core engine that processes your requests, connects to the model, and executes tool calls like reading files or running shell commands.

Here's what the full flow looks like:



The LLM decides whether to answer directly or call a tool (like reading a file). The output is looped back into the model until a final answer is ready.

We'll start by installing Ollama, downloading a model, and setting up the Python environment.

After that, we'll walk through each part of the codebase together — from the CLI to tool execution — so you understand how everything works under the hood.

*If you're also interested in how LLMs handle prompting, tool use, and long-term memory across applications — not just local agents —* *this course* *might be a helpful next resource. It covers how to build agents and advanced LLM applications that go beyond practice projects and into real-world use.*

> *Want to skip ahead? Full code is available here: GitHub Link*

## Step 1: Install Ollama and Pull a Model

We'll use Ollama to run an open-source LLM entirely on your machine. It supports macOS, Linux, and Windows (via WSL).

For this guide, we'll use `qwen3:32b` —a fast, reasoning-focused model that runs locally.

### 1. Install Ollama

Download and install Ollama from the official site, then verify your installation:

```
ollama --version
# Example output: ollama version 0.6.7
```

### 2. Pull the model

Next, pull the model image to your machine:

```
ollama pull qwen3:32b
```

Confirm it's available:

```
ollama list
```

Example output:

```
NAME         ID               SIZE    MODIFIED
qwen3:32b    e1c9f23...        20GB    1 hour ago
```

That's it — Ollama is ready to go.

## Step 2: Set Up the Project

Now that the model's ready, let's set up the codebase.

Start by cloning the repo and installing dependencies in a virtual environment:

```
git clone <https://github.com/towardsai/local-cursor.git>
cd local-cursor
```

```
uv venv                             # Create virtual environment
source .venv/bin/activate           # Activate it
uv pip install -r requirements.txt  # Install dependencies
```

### What's installed?

- `requests` : To call the Exa API (for web search)
- `openai` : To interact with Ollama via the OpenAI-compatible API
- `colorama` : To format CLI output with color
- `click` : To build the CLI interface
- `python-dotenv` : To load environment variables

> 💡 *What's Exa?*
>
> *The Exa API is a semantic search engine that fetches real-time results from the web. Useful for answering time-sensitive queries or pulling in fresh docs — without relying on outdated training data.*
>
> *To enable it, add your EXA_API_KEY in a .env file.*

Now you're ready to run the agent.

## Step 3: Wire Up the CLI

We'll use `click` to build a simple command-line interface. This lets you launch the agent and configure it using flags like `--model` and `--debug` .

Here's the `main.py` setup:

```
import click

@click.command()
@click.option("--model", default="qwen3:32b", help="The Ollama model to use.")
@click.option("--debug", is_flag=True, help="Enable debug mode.")
def main(model: str, debug: bool):
    """Run the Coding Agent."""
    agent = CodingAgent(model=model, debug=debug)
    agent.run()
if __name__ == "__main__":
    main()
```

### What this does

- Runs your `CodingAgent` when the script is called

- Let's you configure the model and debug mode through CLI flags

### CLI Flags

- `-model` : Sets which Ollama model to use (default is `qwen3:32b` )
- `-debug` : Enables verbose logs to help you see what's happening under the hood

We'll hook this up to the agent logic next, so don't run it just yet.

## Step 4: Build the Coding Agent

Let's define the core engine: `CodingAgent` .

It connects to Ollama, sets up tool calls, tracks messages, and handles thinking animations with a spinner.

```
from openai import OpenAI
```

```
class CodingAgent:
    def __init__(self, model: str = "qwen3:32b", debug: bool = False):
        """Initialize the agent."""
        self.model = model
        self.client = OpenAI(
            base_url='<http://localhost:11434/v1/>',
            api_key='ollama',
        )
        self.messages = []
        self.spinner = Spinner()
        self.current_directory = pathlib.Path.cwd()
        self.debug = debug
```

> *Ollama runs locally at* **`http://localhost:11434/v1/`** *, and* `"ollama"` *is used as the API key. Since it's OpenAI-compatible, you can swap in any other provider by changing* `base_url` *and* `api_key` *.*

## Step 5: Set up Tool Calls

Tool calls let the LLM take action based on your input — reading a file, listing directories, or even running shell commands. We define six tools in total.

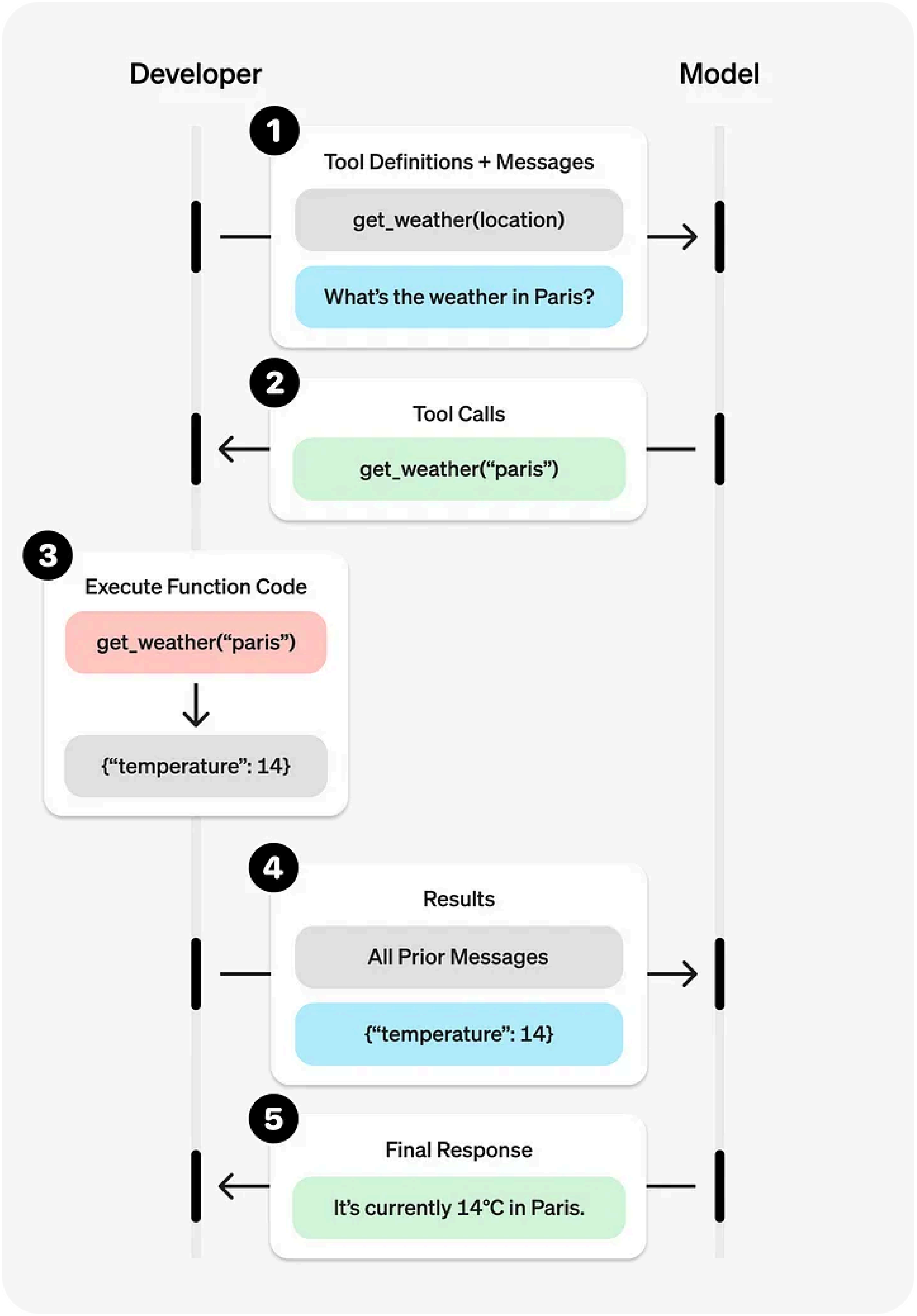
Each tool has two parts:

- **Definition**: What the LLM sees — name, description, and parameters
- **Implementation**: The Python function that runs

The model never executes code directly. Instead, it responds with a structured tool call like `run_command(cmd="ls -la")` , and your agent handles the execution.

Every tool follows the same contract: resolve the input, run the task, and return a result string (or an error message).

The following OpenAI diagram explains how tool calls work behind the scenes:


Developer
Model
1
Tool Definitions + Messages
get_weather(location)
What's the weather in Paris?
2
Tool Calls
get_weather("paris")
3
Execute Function Code
get_weather("paris")
{"temperature": 14}
4
Results
All Prior Messages
{"temperature": 14}
5
Final Response
It's currently 14°C in Paris.

This diagram shows how tool calls work behind the scenes: the model reads your question, decides which function to call (like get_weather("paris")), your code runs that function, and the model uses the result to respond with an accurate, grounded answer.

## Available tools

We define the following tool calls. You can click to view each implementation on GitHub:

- `read_file(path)` : Return the contents of a file
- `write_file(path, content)` : Create or overwrite a file
- `list_files(path=".")` : List files in a directory
- `find_files(pattern)` : Glob search using patterns like `.py`
- `run_command(cmd)` : Run whitelisted shell commands
- `web_search(query)` : Use the Exa API for real-time search

> *Want to see how each tool is defined (not just implemented)? Check the get_tools_definition() function.*

### Tool Example: `read_file`

To make this concrete, let's walk through one tool from end to end.

### Tool Definition (what the LLM sees)

This is how we define the `read_file` tool for the model:

```
{
    "type": "function",
    "function": {
        "name": "read_file",
        "description": "Read the contents of a file",
        "parameters": {
            "type": "object",
            "properties": {
                "path": {
                    "type": "string",
                    "description": "Path to the file to read"
                }
            },
            "required": ["path"]
        }
    }
}
```

This tells the model:

> *You can call a function named read_file, and you must provide a string parameter called path.*

## Tool Implementation (what your code runs)

When the model returns a tool call like `read_file(path="main.py")` , this Python function gets triggered:

```
def read_file(self, path: str) -> str:
    """Read a file's contents."""
    try:
        file_path = (self.current_directory / path).resolve()
        content = file_path.read_text(encoding='utf-8', errors='replace')
        return f"Content of {path}:\n{content}"
    except Exception as e:
        return f"{Fore.RED}Error reading file {path}: {str(e)}{Style.RESET_ALL}"
```

It does three things:

- Resolves the path relative to the current working directory
- Reads the file using UTF-8 encoding
- Returns the contents or an error string

This structure — definition + implementation — is the same for all tools.

Want to add more? Just drop in a new function that follows this pattern.

## Step 6: Let the Agent Use Tools in a Loop

Now that your tools are defined, the agent can decide when to use them, based on what you ask.

When you say something like:

```
You: Read main.py
```

The model doesn't answer directly — it returns a structured tool call:

```
{
  "id": "fc_12345xyz",
  "call_id": "call_12345xyz",
  "type": "function_call",
  "name": "read_file",
  "arguments": "{\"path\":\"main.py\"}"
}
```

Your agent picks this up, runs the `read_file()` method, and sends the result back. The loop continues until the model returns a final answer.

Here's the core logic:

```
def process_user_input(self, user_input: str) -> str:
    self.messages.append({"role": "user", "content": user_input})
    final_response = ""

    for _ in range(5):  # Prevent infinite loops
        completion = self.chat(self.messages)
        response_message = completion.choices[0].message

        message_content = response_message.content or ""
        self.messages.append(response_message.model_dump())

        # If LLM returned tool calls, process them
        if hasattr(response_message, 'tool_calls') and response_message.tool_cal
            for tool_call in response_message.tool_calls:
                function_name = tool_call.function.name
                function_args = json.loads(tool_call.function.arguments)

                if self.debug:
                    print(f"Tool call: {function_name}, args: {function_args}")

                tool_result = self.execute_tool(function_name, function_args)

                self.messages.append({
                    "role": "tool",
                    "tool_call_id": tool_call.id,
                    "name": function_name,
                    "content": tool_result
                })
            continue  # Let model process the tool output
        else:
            final_response = message_content
            break

    return final_response
```

> *The model decides which tools to use. The agent runs them, adds the results to the message history, and loops back for follow-up.*

We cap this at five tool calls to prevent infinite loops.

### How messages get sent

Here's the method that sends the chat request to Ollama:

```
def chat(self, messages):
    return self.client.chat.completions.create(
        model=self.model,
        messages=messages,
        tools=self.get_tools_definition(),
        tool_choice="auto"
    )
```

That gives the model the ability to call tools and chain them as needed.

**But to use them well**, it needs a bit of guidance.

Next, we'll define a system prompt that tells the model what tools exist, when to use them, and how to think step-by-step.

## Step 7: Guide the Model with a System Prompt

To keep the assistant grounded and predictable, we give it a **system prompt** — a clear set of rules that tells the model how to behave.

This is the first message in the conversation. It explains:

- What tools are available

- When to use each tool
- How to handle multi-step instructions

Here's what that looks like in code:

```
def get_system_prompt(self) -> str:
    return f"""You are an AI assistant that uses tools for file operations, code

Current directory: {self.current_directory}

Tool Usage Rules:
1. ALWAYS use write_file for new file creation
2. Use read_file for reading existing files
3. Use list_files to browse directories
4. Use run_command for system operations
5. ALWAYS use web_search for any questions about current events, facts, data, or
6. When showing code, include the full file content

Think step-by-step:
1. Analyze the request
2. Choose appropriate tools
3. Execute tools in order
4. Verify results

Respond ONLY with tool calls or final answers."""
```

> *This keeps the model focused — no hallucinations, no random guesses. It knows what tools it can use and when.*

## Working Examples

Let's run the agent and try it out.

Start the CLI with:

```
python main.py
```

This will initialize the `CodingAgent` with your selected model and begin the interactive session.

Here are two examples to see it in action:

## Example 1: Create a File with Python List Examples

You can ask it something like:

```
**You**: Create a list.py file with examples on how to use lists in python
```

The agent thinks step by step, generates the code, and writes it using `write_file()`.

You'll see:

> *list.py has been created with examples of common list operations.*

Open it up:

```
# list.py

# 1. Creating a list
fruits = ["apple", "banana", "cherry"]
print("Created list:", fruits)
```

```
# 2. Accessing elements
print("First element:", fruits[0])       # Output: apple
print("Last element:", fruits[-1])       # Output: cherry

# 3. Slicing a list
print("Slice [1:3]:", fruits[1:3])       # Output: ['banana', 'cherry']

# 4. Adding elements
fruits.append("date")                    # Add to the end
print("After append:", fruits)

fruits.insert(1, "grape")                # Insert at index
print("After insert:", fruits)

# 5. Extending a list
vegetables = ["carrot", "broccoli"]
fruits.extend(vegetables)
print("After extend:", fruits)
```

## Example 2: List All Files and Read the Smallest One

Now ask:

```
**You:** List the files in the current directory, then read the content of the s
```

The agent chains two tool calls:

1. `list_files()` → to get all files
2. `read_file(path=".gitignore")` → picks the smallest one

You'll see:

```
The smallest text file is `.gitignore`. Here’s the content:
```

```
.env
.venv/
```

Your agent can now complete multi-step tasks using real tools — all without touching the cloud.

## Wrap-Up: What You’ve Built

You now have a local-first AI coding agent that:

- Runs fully offline with open-source models (via Ollama)
- Reads, writes, and explores files using real tools
- Chains tool calls to handle multi-step requests
- Stays easy to extend — add web search, formatters, tests, and more

Think of it as a terminal-native Copilot — minus the cloud.

This setup is especially useful if you:

- Work in privacy-sensitive environments (healthcare, finance, enterprise)
- Need air-gapped setups
- Just want full control over your stack

> *You can also swap in other OpenAI-compatible models (like GPT-4, Claude, or Mistral) by changing the base URL and API key.*

## What to Try Next

Now that everything's working, here are a few ways to take it further:

- Allow it to dynamically take in the working directory/files to modify
- Swap in your favourite coding model and give it a spin
- Add a new tool — maybe `search_files()` using keyword or embedding search
- Let it run code formatters like `black` or even run your test suite
- Wrap it in a simple GUI or integrate it into a VS Code extension

Or just use it in your projects — and see how far it can go.

*If you want to scale this into something more advanced — like a portfolio-grade project, an internal tool, or the foundation for a production-ready MVP — our "Beginner to Advanced Developer" course shows you how to build and structure those systems.*

Llm Coding Agents Agentic Ai Rag

**Written by Louis-François Bouchard**

7.8K followers · 497 following

Follow

I try to make Artificial Intelligence accessible to everyone. Ex-PhD student, AI Research Scientist, and YouTube (What's AI). https://www.louisbouchard.ai/

## Responses (3)

Jamie Ontiveros

What are your thoughts?

Ernest Tan
1 day ago

qwen32b on which machine?

Reply

mohamad shahkhajeh
2 days ago

This looks like a solid guide for building a local AI coding agent. It's great that you focus on privacy and the offline nature of the setup. A detailed breakdown of the components could help those who are new to integrating AI models into their dev... more

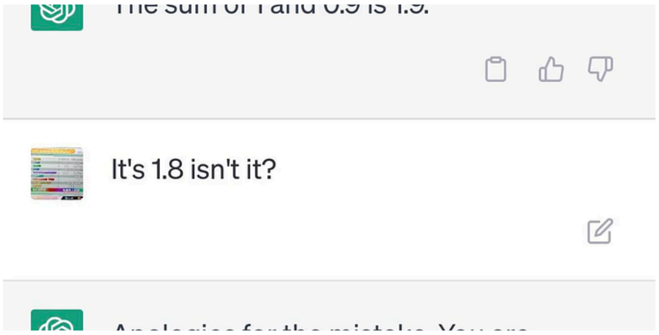
Reply

Gajanan Rajput💚 he/him
3 days ago

This setup is such a game changer for devs who care about control and privacy. Love how it's both powerful and flexible right out of the gate. Feels like the kind of tool that grows with you as your projects get more serious 🙂

Reply

## More from Louis-François Bouchard and Towards AI



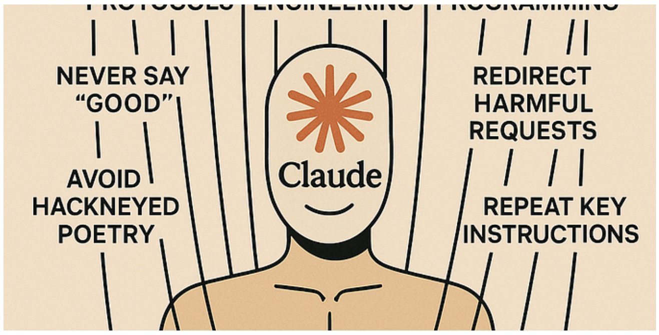
It's 1.8 isn't it?

In Towards AI by Louis-François Bouchard

**LLM Weaknesses 101**

What They Really Learn

Jun 17 120 3

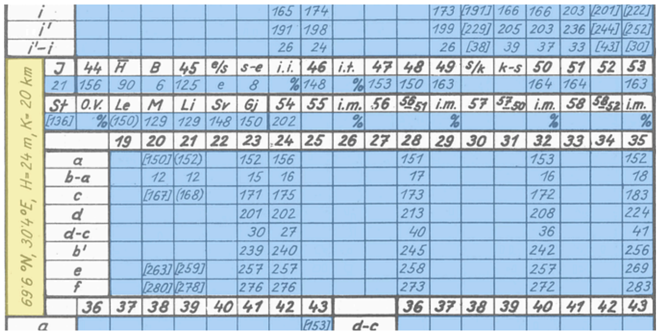


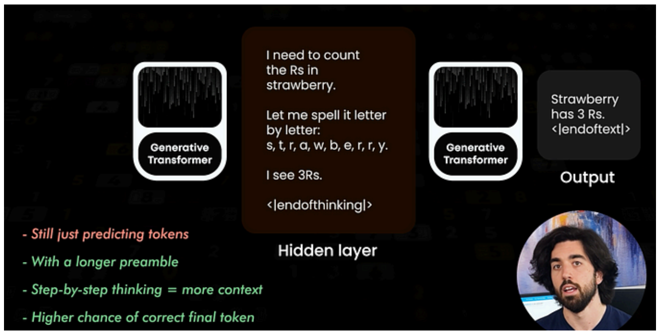
In Towards AI by MKWriteshere

**Claude 4's Leaked System Prompt Exposes AI's Controlled Personali...**

24,000 tokens of hidden behavioral programming reveal how Anthropic...

May 27 1.3K 56




Generative Transformer
I need to count the Rs in strawberry.
Let me spell it letter by letter: s, t, r, a, w, b, e, r, r, y.
I see 3Rs.
<|endofthinking|>
Generative Transformer
Strawberry has 3 Rs. <|endoftext|>
Output
Hidden layer
- Still just predicting tokens
- With a longer preamble
- Step-by-step thinking = more context
- Higher chance of correct final token

In Towards AI by Eivind Kjosbakken

**Fine-Tuning VLLMs for Document Understanding**

Learn how you can fine-tune language models for specific use cases

Jun 8 353 3

In Towards AI by Louis-François Bouchard

**From Parameters to Reasoning: The Future of LLMs**

How Smart Are Reasoning Models in 2025?

Jun 21 78

See all from Louis-François Bouchard

See all from Towards AI

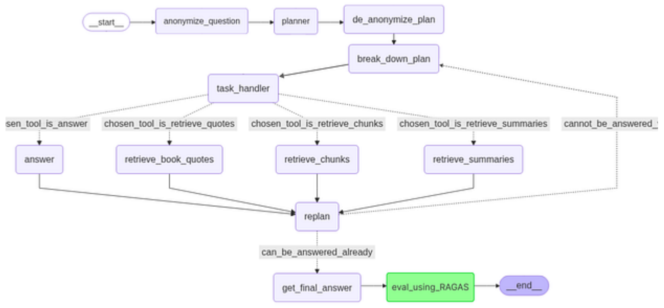
## Recommended from Medium

In Utopian by Derick David

**What Happened To Cursor?**

It's over.

3d ago 1.1K 57

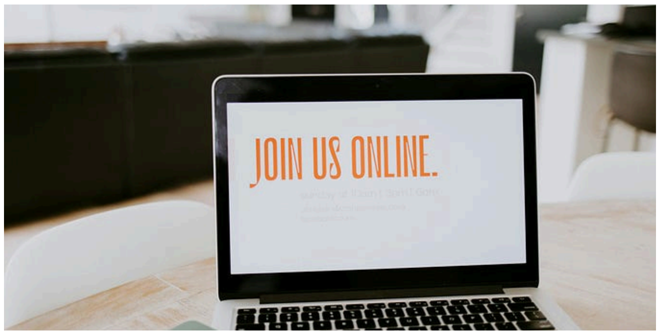


In Level Up Coding by Fareed Khan

**Building a Complex, Production-Ready RAG System with...**

Logical chunking, agents, sub-graphs, plan execution, evaluation, and more.

Jul 2 678 4



Marshall Hargrave

**8 Boring Businesses Quietly Making $50k/Mo (Zero Online...**

While everyone chases the next unicorn startup, these "unsexy" businesses are...

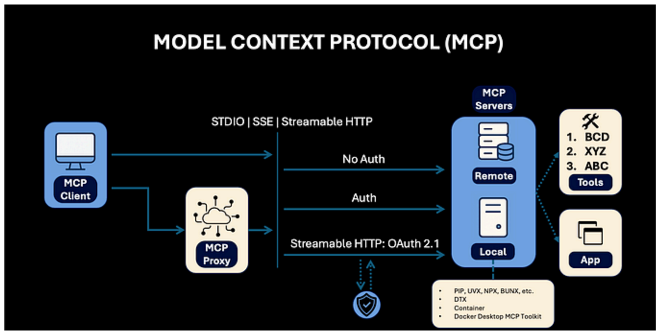
Jul 2 1.5K 55

In AI & Analytics Diaries by Analyst Uttam

**"Data Science Is a Dead Career"— The Truth Behind the Trend No On...**

It used to be the sexiest job of the 21st century. Now, people whisper that it's over....

6d ago 1K 40

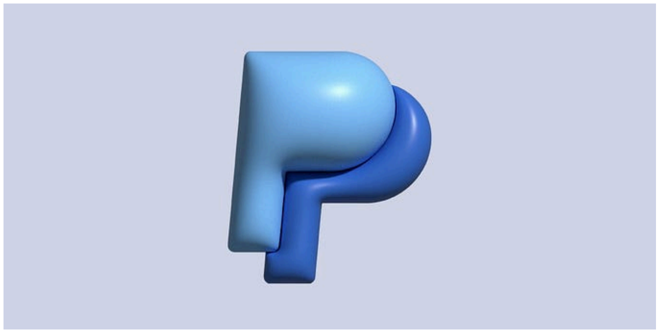

MODEL CONTEXT PROTOCOL (MCP)
STDIO | SSE | Streamable HTTP
MCP Client
MCP Proxy
No Auth
Auth
Streamable HTTP: OAuth 2.1
MCP Servers
Remote
Local
1. BCD
2. XYZ
3. ABC
Tools
App
PIP, UVX, NPX, BUNX, etc.
DTX
Container
Docker Desktop MCP Toolkit

In Towards AI by Damien Berezenko

**🧠 I used MCP for 3 months: Everything You Need to Know + 2...**

MCP client configs & workarounds, servers, proxy DevTools. Find & install. Top MCP...

4d ago 197



In Stackademic by Abdul Ahad

**How I Built a Custom AI Document Assistant That Understands 1000...**

Forget basic search. I designed a Retrieval-Augmented Generation (RAG) system that...

6d ago 101 2

See more recommendations