Building AI Agents from Scratch

Week 12: CS 203 - Software Tools and Techniques for AI

Prof. Nipun Batra

IIT Gandhinagar

Where We Are in the Course

```
Week 7-8:   Build and tune the model      → Is it good?
Week 9:     Reproducibility                → Can someone re-run it?
Week 10:    Data drift                     → Is it STILL good?
Week 11:    Profiling & Quantization       → Is it FAST and SMALL?
Week 12:    Agents from Scratch            → Can it DO things?   ← today
```

Weeks 7-11 were about building and shipping a model that *answers questions*.

Today we build something that **takes actions**.

A Simple Question, A Hard Answer

You ask an LLM: *"What's the weather in Gandhinagar right now?"*

The LLM replies:

> "I don't have access to real-time data, but Gandhinagar typically experiences temperatures around 35-40°C in April..."

It *guessed*. It didn't *check*. Why?

An LLM is just a text predictor. It can only generate the next word.
It can't open a browser, call an API, run code, or check a database.

What If It *Could* Call an API?

Imagine the LLM could say:

> "I need to check the weather. Let me call the weather API."

Then it produces this:

```
{"tool": "get_weather", "arguments": {"city": "Gandhinagar"}}
```

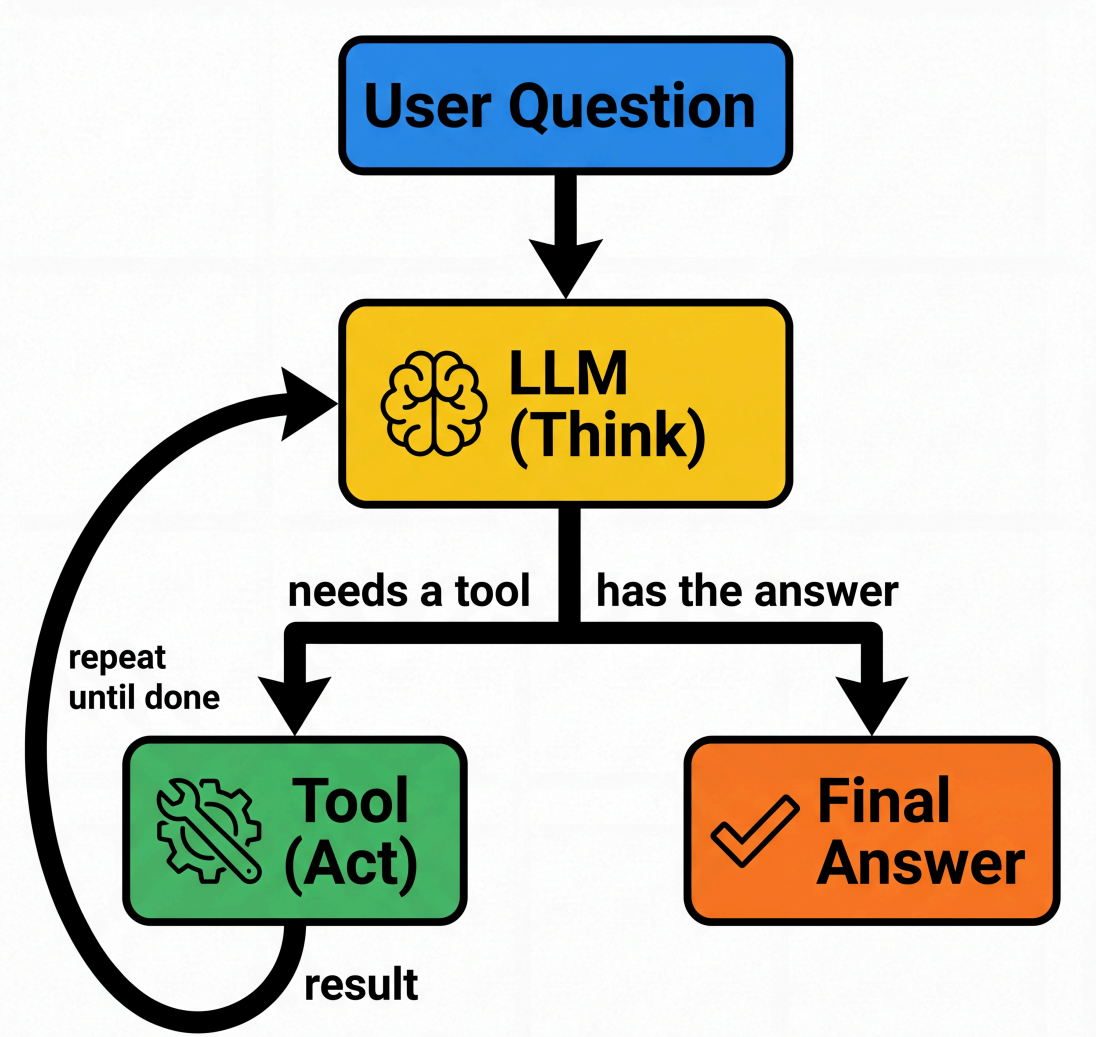
Your code executes that, gets back `{"temp": 38, "condition": "Sunny"}`, and feeds it to the LLM, which then says:

> "It's currently 38°C and sunny in Gandhinagar."

That's an agent. An LLM that can *use tools*.

What Is an Agent?

An agent = LLM + tools + a loop.



You Already Use Agents Every Day

App	What you say	What the agent *does*
Google Assistant	"Set a timer for 10 minutes"	Calls the clock API
Siri	"Text Mom I'll be late"	Calls the messaging API
ChatGPT with browsing	"Summarize today's news"	Calls the Bing search API
Claude Code	"Fix the failing test"	Reads files, edits code, runs pytest
GitHub Copilot	"Add error handling here"	Reads context, writes code inline

In every case: an LLM **decides** what action to take, your device **executes** it, and the result goes **back to the LLM** for the next step.

Deep Dive: What Does Claude Code Actually Do?

You type: **"Fix the failing test in test_api.py"**

Claude Code doesn't magically know the answer. Here's what the agent loop does:

Step	Action	Tool used
1	Read `test_api.py` to understand the test	`Read file`
2	Run `pytest test_api.py` to see the error	`Bash (shell)`
3	Read the error traceback	`Read output`
4	Read `app.py` (the file under test) to find the bug	`Read file`
5	Edit line 42 of `app.py` to fix the bug	`Edit file`
6	Run `pytest test_api.py` again to verify	`Bash (shell)`
7	Tests pass → report success to user	*No tool — done*

Seven steps, four different tools, zero human intervention. The LLM decided what to do at every step based on what it saw.

Exercise: Trace the Agent Loop (2 minutes)

Pick one of these prompts and write down what tools an agent would need and what steps it would take. Share with your neighbour.

Prompt	Think about it...
"Book me a flight from Ahmedabad to Delhi under ₹5000"	What APIs? What order?
"Summarize the top 3 news stories about AI today"	Search, scrape, summarize?
"Generate a bar chart of India's GDP from 2015 to 2024"	Search data, write code, run it?
"Review this pull request and leave comments"	Read diff, read code, write comments?

There's no single right answer — the fun part is thinking about which tools and in what order. This is what AI engineers design every day.

Three Things That Make an Agent

Piece	What it does	Analogy
LLM	Thinks, reasons, decides what to do next	The brain
Tools	Functions the LLM can call (calculator, search, code runner)	The hands
Loop	Keeps going until the task is done	The work ethic

Without tools, the LLM can only *talk*. With tools, it can *do*.

Without a loop, it can do *one thing*. With a loop, it can do *multi-step tasks* — like "find the cheapest flight, then book it."

The "Restaurant" Analogy

Without tools (plain LLM):

> You ask the chef: "What's a good recipe for biryani?"
> The chef tells you the recipe from memory. Helpful, but you still need to cook it yourself.

With tools (agent):

> You ask the chef: "Make me biryani."
> The chef *checks the fridge* (tool: inventory lookup), *orders missing spices* (tool: delivery API), *sets the oven* (tool: oven control), and *serves you the biryani*.

The chef (LLM) still has the knowledge. The tools let it **act on that knowledge**.

Part 1: How Function Calling Works

The model describes what it wants. You execute it.

The Key Insight

Function calling is NOT the model running code. The model never executes anything. Here's what happens:

Step	Who does it	What happens
1	**You**	Hand the model a *menu* of available tools
2	**Model**	Reads the question, *decides* which tool to call
3	**Model**	Outputs a *structured request*: `{"name": "get_weather", "args": {"city": "Gandhinagar"}}`
4	**You**	Execute the function, get the result
5	**You**	Feed the result *back* to the model
6	**Model**	Writes the final answer using the real data

The model is a **decision maker**. You are the **executor**.

Defining a Tool = Writing a Menu Item

Every tool has two parts: a Python function and a description (JSON schema).

```
# The function (what actually runs)
def get_weather(city: str) -> str:
    data = {"Gandhinagar": {"temp_c": 38, "condition": "Sunny"}, ...}
    return json.dumps(data.get(city, {"error": "unknown city"}))

# The description (what the model reads to decide)
weather_tool = {
    "type": "function",
    "function": {
        "name": "get_weather",
        "description": "Get current weather for a city.",
        "parameters": {
            "type": "object",
            "properties": {"city": {"type": "string"}},
            "required": ["city"]
        }
```

The model never sees the function code — only the description.

Part 2: Four Use Cases

Each one shows a different kind of tool and why it matters.

Use Case 1: Calculator — LLMs Can't Do Math

Ask an LLM *"What is 4729 times 8314?"* and it will confidently give the wrong answer. Token prediction is not arithmetic.

Fix: a `calculate(expression)` tool that calls Python's `eval()`.

What the user asks	What the model calls
"What's 18% tip on ₹2,450?"	`calculate("2450 * 0.18")`
"If I invest ₹10,000 at 8% for 5 years?"	`calculate("10000 * (1.08 ** 5)")`
"Average of 82, 91, 76, 88"	`calculate("(82+91+76+88) / 4")`
"What's the square root of 2?"	`calculate("2 ** 0.5")`

Every time an LLM needs *exact* arithmetic, it should call a tool instead of computing in its "head."

Use Case 2: Weather — LLMs Have No Real-Time Data

An LLM's training data has a cutoff date. It can't tell you today's temperature any more than a textbook can.

Fix: a `get_weather(city)` tool backed by mock data (or a real API).

Ask: *"Should I carry an umbrella in Bangalore today?"*

Model calls `get_weather("Bangalore")` → `{"condition": "Rainy", "humidity": 65}` → *"Yes — it's rainy with 65% humidity."*

The model used **actual data** instead of guessing from training statistics.

Use Case 3: Unit Converter — Precision Matters

Is 100°F hot or cold? Most people need to convert to answer that.

Fix: a `convert_units(value, from_unit, to_unit)` tool with 10 conversion pairs.

What the user asks	What the model calls
"I weigh 70 kg — in pounds?"	`convert_units(70, "kg", "pounds")`
"Convert 100°F to Celsius"	`convert_units(100, "fahrenheit", "celsius")`
"How far is 5 miles in km?"	`convert_units(5, "miles", "km")`

Use Case 4: Course Notes Search — Your Own Knowledge Base

"What week did we cover data drift?" — the LLM doesn't know *your* course.

Fix: a `search_notes(query)` tool backed by a dict of CS 203 topics.

```
topics = {
    "data drift":   "Week 10: KS test, PSI, distribution shift",
    "profiling":    "Week 11: cProfile, timeit, bottlenecks",
    "quantization": "Week 11: INT8, ONNX, model compression",
    "docker":       "Week 10: containerization, Dockerfiles",
    "agents":       "Week 12: tool calling, Gemma 4, agent loop",
    ...
}
```

This is a **tiny RAG system**: retrieval (search the dict) + generation (model writes a natural-language answer from the result).

All Four Tools Together — The Model Picks the Right One

User question	Tool the model picks
"What's 17 squared?"	`calculate("17 ** 2")`
"Is it raining in Mumbai?"	`get_weather("Mumbai")`
"Convert 100°F to Celsius"	`convert_units(100, "fahrenheit", "celsius")`
"When did we learn about Docker?"	`search_notes("docker")`
"What is the capital of France?"	*No tool* — answers from memory

The model reads the descriptions and **chooses which tool fits the question**, or answers directly if no tool is needed. You'll try all of these in the notebook.

Part 3: The Agent Loop

The only architecture diagram you need to remember.

The Agent Loop in Pseudocode

```
messages = [user's question]

repeat (up to max_steps):
    response = model.generate(messages, tools=...)

    if response has NO tool call:
        return response as final answer     ← done!

    for each tool_call in response:
        result = execute(tool_call)         ← YOUR code runs
        append result to messages

    go back to top of loop                  ← model sees the result
```

That's it. ~10 lines of real Python. You'll write it yourself in the notebook.

Multi-Step Example

"What's the square root of 144 plus the temperature in Mumbai in Fahrenheit?"

Step	Model decides	Tool called	Result
1	"I need sqrt(144)"	`calculate("144 ** 0.5")`	`12.0`
2	"I need Mumbai's temp"	`get_weather("Mumbai")`	`32°C`
3	"Convert 32°C to °F"	`convert_units(32, "celsius", "fahrenheit")`	`89.6°F`
4	"Add them"	`calculate("12.0 + 89.6")`	`101.6`
5	*No tool needed*	—	"The answer is 101.6"

Four tool calls, three different tools, one coherent answer. The model planned what to do — you just executed each step.

Part 4: Where This Is Going

From toy demos to real-world impact.

Agents in the Wild (2025-2026)

Agent	What it does	Tools it uses
Claude Code	Writes, edits, and tests code from your terminal	File read/write, bash, grep, git
Cursor / Windsurf	AI-powered code editors	File system, LSP, terminal
Devin	Autonomous software engineer	Browser, terminal, code editor
Perplexity	Search engine with citations	Web search, web scrape
Google Deep Research	Multi-step research reports	Search, summarize, cite
OpenAI Operator	Uses websites on your behalf	Browser automation

Every one of these is the same pattern you just learned:
LLM + tools + loop. The tools are more powerful, the models are bigger,
but the architecture is identical.

MCP — The Universal Tool Standard

Today you defined tools as Python dicts. But what if you want to share tools across different LLMs and applications?

MCP (Model Context Protocol) is an open standard from Anthropic that lets anyone publish tools that any agent can use:

One tool definition, many LLMs. Like USB-C for AI tools.

The Agent Stack — A Mental Model

```
Layer 4:  Application      Claude Code, Cursor, Devin, your app
                           ↑
Layer 3:  Agent framework  The loop + memory + planning
                           ↑
Layer 2:  Tools            Calculator, search, file I/O, APIs
                           ↑
Layer 1:  LLM              Gemma 4, Claude, GPT, Llama
                           ↑
Layer 0:  Hardware          Your laptop, Colab T4, cloud GPU
```

Today you built layers 1-3 from scratch. Layer 4 is what you build when you put it all together into a product.

The entire field of "AI engineering" in 2025-2026 is about making this stack reliable, fast, and safe.

Guardrails: When Agents Go Wrong

Agents can be powerful but also unpredictable. Real-world agents need **guardrails**:

Risk	Example	Guardrail
Infinite loop	Agent keeps calling tools forever	`max_steps` parameter
Wrong tool	Agent calls `delete_file` when it meant `read_file`	Require user confirmation for dangerous tools
Hallucinated args	`get_weather("Narnia")`	Validate arguments before execution
Cost explosion	Agent makes 1000 API calls	Budget / rate limiting

Our `max_steps=5` parameter is the simplest guardrail. Production agents add human-in-the-loop confirmation, sandboxing, and cost limits.

Summary

Key Takeaways

1. **An agent = LLM + tools + loop.** The LLM *decides*, you *execute*.
2. **Function calling** is how the model requests tool use — it produces structured JSON, not free-form text.
3. **Tools are just Python functions** with a description. The model reads the description and decides when to use each one.
4. **The agent loop** is ~10 lines: generate → check for tool call → execute → feed result back → repeat.
5. **Gemma 4 E2B** runs on a free Colab T4 with 4-bit quantization and has native tool-calling support.
6. **This is the architecture behind Claude Code, Cursor, Devin, Perplexity**, and every other AI agent you use. Same pattern, bigger tools.

What to Try in the Notebook

Task	What you'll do
Load Gemma 4	Run a 4-bit model on free Colab T4
Single tool call	Define a calculator tool, watch the model use it
Multi-tool agent	Give the model 4 tools, ask compound questions
Build your own tool	Write a new function, add it to the agent
Multi-step reasoning	Ask questions that need 3+ tool calls

Open the notebook in Colab

Questions?

LLM + Tools + Loop = Agent.