Lesson 9 · Ollama + Function Calling

PDF: this lesson · **Install (Linux · macOS · Windows):** guide · PDF

> **Part of local-ai-lab** - a hands-on course for building local AI.

> **Course home:** https://nikolareljin.github.io/local-ai-lab/ **Source:** https://github.com/nikolareljin/local-ai-lab

> **Lessons:** 1 · RAG → 2 · MCP → 3 · Hybrid retrieval → 4 · RAG safety → 5 · RAG evaluation → 6 · Repo assistant → 7 · LangChain → 8 · LangGraph → **9 · Ollama tools (you are here)** → 10 · Semantic Kernel → 11 · Bedrock Agents → 12 · Google ADK → ... → 15 · Docs from changes

> **Status: planned.** Outline below; full published slideshow lesson + step-by-step coming later. **Runs 100% locally** (Ollama, no cloud). Star the repo to follow along.

The idea

A chat model that can only talk is limited. **Function calling** (a.k.a. tool use) lets the model decide to call **your** functions - search documents, do math, hit an API - and use the results to answer. In this lesson you give a **local Ollama model** real tools, fully offline.

This is the bridge between Lesson 1's RAG and Lesson 2's MCP: same `search_docs` capability, but now the **model** chooses when to call it.

What you'll learn

- **Tool schemas** - describing functions as JSON so the model knows when and how to call them
- **The tool-call loop** - send tools → model emits `tool_calls` → you execute → return results → repeat
- **Local function calling with Ollama** - `POST /api/chat` with a `tools` array (models like `llama3.1`, `qwen2.5`, `mistral-nemo` support it)
- **Grounded tools** - wiring the Lesson 1 retriever in as a `search_docs` tool that returns cited text
- **Guardrails** - validating arguments, limiting iterations, handling a model that loops

The design we'll build

```
import requests
from localrag.config import load_config
from localrag.engine import get_retriever

TOOLS = [{
   "type": "function",
   "function": {
       "name": "search_docs",
```

```
        "description": "Search the user's local documents for relevant passages.",
        "parameters": {
            "type": "object",
            "properties": {"query": {"type": "string"}},
            "required": ["query"],
        },
    },
}]

def run_tool(name, args):
    if name == "search_docs":
        hits = get_retriever(load_config()).search(args["query"], 5)
        return "\n\n".join(f"[{h['source']}:{h['page_number']}] {h['text']}" for h in hits)
    return "unknown tool"

def chat(messages):
    # POST {OLLAMA_URL}/api/chat with model + messages + tools
    # if the reply has message.tool_calls: run each, append results, call chat() again
    # else: return the final grounded answer
    ...
```

Builds on

Concept	From
`get_retriever()` / `search_docs`	Lesson 1
the same tool exposed over a protocol	Lesson 2 (MCP)
the model deciding to call tools	**Lesson 9 (this one)**

Prerequisites

Lesson 1, plus Ollama installed with a tool-capable model pulled (`ollama pull llama3.1`).

Next lesson

Lesson 10 · Microsoft Semantic Kernel (C#) →
