Function Calling with Local LLMs: Tools, Agents, and Structured Output

February 11, 2026 · by Mark Bartlett

Download this guide as PDF

Quick Answer: Qwen 2.5 7B is the best function-calling model for most local setups — it hits 0.933 F1 for tool selection at just 5-6GB VRAM. Ollama's tools API handles the plumbing: you define functions, the model outputs structured tool calls, you execute them, feed results back. Local models now match cloud API accuracy for single tool calls. Multi-step chains and knowing when not to call a tool are where smaller models still struggle. Start with Ollama + Qwen 2.5 7B, graduate to 14B for anything agentic.

📚 **Related:** Structured Output from Local LLMs · llama.cpp vs Ollama vs vLLM · Qwen Models Guide · Best Coding Models

Cloud APIs have had function calling for years. You give GPT-4 a list of tools, it decides which one to call, you execute it, feed the result back. It's how every AI agent works under the hood.

Local models can do this now. Ollama added tool support, llama.cpp has native function calling handlers, and models like Qwen 2.5 match GPT-4's accuracy at picking the right tool. The gap isn't whether it works — it's knowing which models to use, which failure modes to watch for, and how to structure the agentic loop so it doesn't spiral.

This guide covers the practical side: working code, model recommendations, and the patterns that hold up when you move past demos.

What Function Calling Actually Is

Function calling is not the model running code. The model never executes anything. It writes a structured request asking you to run a function on its behalf.

The flow works like this:

1. You send the model a message plus a list of available tools (name, description, parameters)
2. The model decides whether to call a tool or respond with text
3. If it wants a tool, it outputs a JSON object: `{"name": "get_weather", "arguments": {"city": "Tokyo"}}`

4. Your code executes the real function and gets a result
5. You send the result back to the model as a “tool” message
6. The model uses the result to form a natural-language response

The model’s only job is to produce the right JSON. Your code handles everything else. This separation is what makes it work — and what makes it fail, because the model can produce JSON that’s syntactically valid but semantically wrong.

Function calling is different from structured output. Structured output means “give me valid JSON in a specific schema.” Function calling means “choose a tool from this list and provide the right arguments.” Function calling uses structured output under the hood, but adds the decision layer of which tool and when.

Which Models Support It

Not every local model can do function calling. The model needs to be trained on tool-use data with the right special tokens and chat template. These are the ones worth using:

Model	Size	VRAM (Q4)	Tool Accuracy	Best For
Qwen 2.5 7B	7B	~5-6 GB	0.933 F1	Default pick. Best accuracy per GB.
Qwen 2.5 14B	14B	~9-10 GB	0.971 F1	Near-GPT-4 accuracy. Multi-step chains.
Llama 3.1 8B	8B	~5-6 GB	89% overall	Native tool calling. Good all-rounder.
Llama 3.3 70B	70B	~42 GB	94%+	Best accuracy if you have the VRAM.
Mistral 7B v0.3	7B	~5 GB	Good	Fastest inference. 457 tok/s.
Mistral Nemo 12B	12B	~7-8 GB	Good	128K context. Solid mid-range.
Mistral Small 24B	24B	~15 GB	Strong	Best agentic capabilities at this size.
Hermes 2 Pro 8B	8B	~5-6 GB	Good	XML-based tool format. vLLM/SGLang support.
Mixtral 8x7B	56B (13B active)	~24 GB	88% overall	Expert routing. Good multilingual.
Functionary v3	8B	~5-6 GB	Strong	Purpose-built for function calling.

The default recommendation: Start with Qwen 2.5 7B. In Docker's practical evaluation, Qwen 3 14B hit 0.971 F1 for tool selection — 0.003 behind GPT-4's 0.974. The 7B variant at 0.933 is still excellent and runs on an 8GB GPU.

Llama 3.1 8B is the runner-up. Meta baked tool calling into the training, including three built-in tools (web search, Wolfram Alpha, code interpreter) that activate when you put `Environment: ipython` in the system prompt. It uses special tokens like `<|python_tag|>` and an `ipython` role for tool results.

Hermes models use a different format: XML tags (`<tool_call>`, `<tool_response>`) instead of the OpenAI-style JSON. This works well with vLLM and SGLang's automatic tool parsers but can confuse tools that expect the OpenAI format. Qwen 2.5 actually borrowed the Hermes format, so they're compatible.

How It Works in Ollama

Ollama's tools API follows the OpenAI format. If you've used OpenAI's function calling, the interface is almost identical.

curl Example

```
curl http://localhost:11434/api/chat -s -d '{
  "model": "qwen2.5:7b",
  "messages": [
    {"role": "user", "content": "What is the weather in Tokyo?"}
  ],
  "stream": false,
  "tools": [
    {
      "type": "function",
      "function": {
        "name": "get_weather",
        "description": "Get current weather for a city",
        "parameters": {
          "type": "object",
          "properties": {
            "location": {
              "type": "string",
              "description": "City name, e.g. Tokyo"
            },
            "unit": {
              "type": "string",
              "enum": ["celsius", "fahrenheit"]
            }
          },
          "required": ["location"]
        }
      }
    }
  ]
}'
```

The model responds with `tool_calls` instead of `content`:

```
{
  "message": {
    "role": "assistant",
    "content": "",
    "tool_calls": [
      {
        "function": {
          "name": "get_weather",
          "arguments": {
            "location": "Tokyo",
            "unit": "celsius"
          }
        }
      }
    ]
  }
}
```

Your code executes `get_weather("Tokyo", "celsius")`, then sends the result back as a `tool` role message.

Python with Auto-Schema

The Ollama Python SDK has a nice trick: pass Python functions directly and it builds the tool schema from the function signature and docstring. No manual JSON schema needed.

```
import ollama
import json
```

```python
def get_weather(location: str, unit: str = "celsius") -> str:
    """Get current weather for a city.

    Args:
        location: City name, e.g. Tokyo
        unit: Temperature unit, celsius or fahrenheit
    """
    # Your real API call goes here
    return json.dumps({"temperature": 22, "unit": unit, "condition": "clear"})

response = ollama.chat(
    model="qwen2.5:7b",
    messages=[{"role": "user", "content": "What's the weather in Tokyo?"}],
    tools=[get_weather],  # Pass the function directly
)

if response.message.tool_calls:
    for call in response.message.tool_calls:
        print(f"Model wants to call: {call.function.name}")
        print(f"With arguments: {call.function.arguments}")
```

The SDK inspects your type hints and Google-style docstring to create the schema automatically. This is the fastest way to prototype.

Tool Calling vs JSON Mode

These are different features that people mix up:

Feature	What It Does	When to Use
`tools` parameter	Model decides whether to call a tool and outputs structured tool calls	When the model needs to take actions (API calls, DB queries, calculations)
`format: "json"`	Forces the model to output valid JSON (no schema)	When you need raw JSON output, not tool decisions
`format: {schema}`	Forces output to match a specific JSON schema	When you need structured data extraction. See the structured output guide.

Tool calling includes the decision: should I call a tool, and if so, which one? JSON mode just forces the output format. They solve different problems.

How It Works in llama.cpp

If you're running llama.cpp directly instead of Ollama, function calling works through the server's OpenAI-compatible API.

Starting the Server

```
llama-server --jinja -fa \
  -hf bartowski/Qwen2.5-7B-Instruct-GGUF:Q4_K_M
```

The `--jinja` flag enables chat template processing (required for tool calling). The `-fa` flag enables flash attention.

Tool Calling Request

```
curl http://localhost:8080/v1/chat/completions \
  -H "Content-Type: application/json" \
  -d '{
    "model": "qwen2.5",
    "messages": [
      {"role": "user", "content": "What is 42 * 17?"}
    ],
    "tools": [
      {
        "type": "function",
        "function": {
          "name": "calculator",
          "description": "Evaluate a math expression",
          "parameters": {
            "type": "object",
            "properties": {
              "expression": {
                "type": "string",
                "description": "Math expression, e.g. 42 * 17"
              }
            },
            "required": ["expression"]
          }
        }
      }
    ]
  }'
```

llama.cpp has native handlers optimized for specific models: Llama 3.x, Qwen 2.5, Hermes, Mistral Nemo, Functionary, and Command R7B. Other models fall back to a generic handler that still works but may be less reliable.

GBNF Grammars for Custom Constraints

llama.cpp's grammar system constrains output at the token level. The model cannot generate tokens that violate the grammar. This is the same mechanism Ollama uses under the hood when you pass a JSON schema.

For tool calling, you rarely need to write grammars manually — the server handles it. But if you want a custom output format that isn't standard JSON, GBNF gives you token-level control:

```
llama-cli -m model.gguf \
  --grammar-file grammars/json.gbnf \
  -p 'Output the result as JSON:'
```

When to Use llama.cpp Over Ollama

Ollama wraps llama.cpp and adds model management, easier setup, and a friendlier API. For most tool-calling use cases, Ollama is simpler.

Use llama.cpp directly when you need:

- Custom GBNF grammars beyond JSON schemas
- More control over sampling parameters during tool calls
- To avoid Ollama's overhead in high-throughput scenarios
- A specific model that Ollama doesn't support yet

Building an Agentic Loop

A single tool call is easy. The hard part is the loop: the model calls a tool, gets a result, decides whether to call another tool, and eventually produces a final answer. Here's a pattern that handles the failure modes.

```
import ollama
import json

# --- Define your tools ---

def search_products(query: str, max_results: int = 5) -> str:
    """Search the product database.

    Args:
        query: Search term
        max_results: Maximum results to return
    """
    # Simulated — replace with real DB call
    return json.dumps({
        "results": [
            {"name": "Widget Pro", "price": 29.99, "in_stock": True},
            {"name": "Widget Basic", "price": 9.99, "in_stock": False},
        ]
    })

def calculate(expression: str) -> str:
    """Evaluate a math expression safely.

    Args:
        expression: Math expression like '29.99 * 1.08'
    """
    allowed = set("0123456789+-*/.() ")
    if not all(c in allowed for c in expression):
        return json.dumps({"error": "Invalid characters"})
    try:
        return json.dumps({"result": round(eval(expression), 2)})
    except Exception as e:
        return json.dumps({"error": str(e)})

# --- Tool registry ---
tools = [search_products, calculate]
tool_map = {
    "search_products": search_products,
    "calculate": calculate,
}

# --- Agentic loop ---
def run_agent(user_message: str, model: str = "qwen2.5:7b", max_steps: int = 10):
    messages = [{"role": "user", "content": user_message}]

    for step in range(max_steps):
        response = ollama.chat(model=model, messages=messages, tools=tools)
        # No tool calls — model is done
        if not response.message.tool_calls:
            return response.message.content

        messages.append(response.message)

        for call in response.message.tool_calls:
            func_name = call.function.name
            func_args = call.function.arguments

            # Guard: hallucinated function name
            if func_name not in tool_map:
                messages.append({
                    "role": "tool",
                    "content": json.dumps({"error": f"Unknown function: {func_name}"}),
                    "tool_name": func_name,
                })
                continue

            # Guard: wrong argument types
            try:
                result = tool_map[func_name](**func_args)
            except TypeError as e:
                result = json.dumps({"error": f"Bad arguments: {e}"})
            except Exception as e:
                result = json.dumps({"error": f"Failed: {e}"})

            messages.append({
                "role": "tool",
                "content": result,
                "tool_name": func_name,
            })

    return "Agent hit max steps without producing a final answer."

# --- Use it ---
answer = run_agent(
    "Find the Widget Pro and tell me the price with 8% sales tax."
)
print(answer)
```

The key details that make this work in practice:

- **Max iterations.** Without a cap, a confused model loops forever. 10 is a reasonable default.
- **Unknown function guard.** Models hallucinate function names, especially smaller ones. Check the name against your registry before executing.

- **TypeError catch.** Models pass wrong argument types or miss required fields. Return a clear error so the model can retry.
- **The loop exits when `tool_calls` is empty.** That's how the model signals "I have enough information to answer."

Local vs Cloud Function Calling

The accuracy gap has mostly closed. The latency gap hasn't.

Accuracy

Docker ran a practical evaluation pitting local models against cloud APIs on real tool-selection tasks:

Model	F1 Score (Tool Selection)	Type
GPT-4	0.974	Cloud
Qwen 3 14B	**0.971**	Local
Claude 3 Haiku	0.933	Cloud
Qwen 3 8B	**0.933**	Local

Qwen 14B is within 0.003 of GPT-4. For single tool calls with clear intent, local models match cloud.

Latency

	Cloud API	Local (RTX 3090)
Single tool call	3-5 sec	5-15 sec
Multi-step (3 tools)	10-15 sec	30-60 sec
Tokens/sec	80-90 (GPT-4o)	50-112 (7B-8B Q4)

Local is slower, mostly because of lower tokens/sec on consumer hardware. The gap shrinks with faster GPUs and smaller models.

Where Local Wins

- **Privacy.** Your function arguments never leave your machine. If you're querying internal databases or processing customer data, that matters.
- **Cost.** Zero per-call cost after hardware. GPT-4 function calling at scale gets expensive.
- **No rate limits.** No throttling during peak hours. No surprise API changes.
- **Offline.** Works on air-gapped networks, planes, bad WiFi.

Where Cloud Still Wins

- **Multi-step reasoning.** GPT-4 and Claude handle 5+ step tool chains reliably. Local 7B models start losing coherence after 2-3 steps.
- **Knowing when NOT to call a tool.** Local models, especially smaller ones, tend to call tools eagerly — even for questions they can answer directly. Docker's evaluation flagged this as the biggest weakness.
- **Parallel tool calls.** Some cloud APIs support calling multiple tools in one turn. Fewer local models handle this.
- **Recovery from errors.** Cloud models self-correct better when a tool call fails. Local models often repeat the same broken call or enter a degenerate loop.

For single-tool use cases (weather lookup, database query, calculator), local is ready. For complex agentic workflows with branching decisions, larger models (14B+) or cloud APIs are still more reliable.

Common Failures and How to Fix Them

These are the issues you'll hit in practice, not in demos.

Hallucinated Function Names

The model invents a function that doesn't exist in your tool list. This happens more with smaller models and when you have many tools defined.

Fix: Always validate `tool_call.function.name` against your tool registry before executing. Return an error message so the model can try again.

Eager Tool Invocation

The model calls a tool when it shouldn't — like calling `search_web` in response to "Hello, how are you?" Small models are worst at this.

Fix: Add explicit instructions in your system prompt: "Only call tools when you need external data you don't already have." Validate whether the user's question actually needs a tool before executing. Ollama doesn't support `tool_choice` yet, so you can't force "auto" behavior at the API level.

The Bad-State Loop

After a failed tool call, the model repeats the user's input, produces empty responses, or keeps calling the same broken function. This happens across Llama 3, Hermes, and Qwen models.

Fix: Set max iterations on your agentic loop. If the model produces an empty response or repeats itself, break the loop and return a fallback. Don't let it spin.

Context Pressure with Many Tools

Each tool definition costs 50-150 tokens depending on how detailed the description and parameters are. Ten tools can consume 1,000+ tokens of your context window before the conversation starts.

Fix: Keep tool counts under 5-10 for 7B models. Use concise descriptions. For large tool sets, consider injecting only relevant tools based on the user's message rather than loading everything every turn.

Wrong Parameters

The model passes the right function name but wrong argument types, missing required fields, or values that don't match the `enum`.

Fix: Validate arguments against the schema before executing. Return specific error messages ("Missing required parameter: location") rather than generic errors. The model uses error messages to correct its next attempt.

KV Cache Quantization

This one is subtle. If you're running llama.cpp with aggressive KV cache quantization (`-ctk q4_0`), tool calling accuracy degrades. The precision loss in the attention cache affects the model's ability to track tool schemas.

Fix: Use Q8 or higher for the KV cache when doing tool calling. The VRAM savings from Q4 KV cache aren't worth the reliability hit.

Grammar Guarantees Structure, Not Semantics

Ollama's `format` parameter and llama.cpp's grammar enforcement guarantee valid JSON. But the model can still fill valid JSON with wrong data — `{"temperature": 999}` is valid JSON with a nonsensical value.

Fix: Validate the returned values in your code, not just the structure. Treat tool call arguments the same way you'd treat user input: sanitize and bounds-check.

The Bottom Line

Function calling with local LLMs works. Not "sort of works with caveats" — actually works, with accuracy matching cloud APIs for single tool calls.

The setup in three steps:

```
# 1. Install Ollama and pull a model with tool support
ollama pull qwen2.5:7b

# 2. Use the tools API (curl, Python, or any OpenAI-compatible client)
# 3. Build the agentic loop with guards (see code above)
```

For simple tool use (one function, clear intent): Qwen 2.5 7B on 8GB VRAM. It just works.

For multi-step agents (chaining tools, branching logic): Qwen 2.5 14B on 12GB, or Mistral Small 24B on 24GB. Smaller models lose coherence after 2-3 steps.

For maximum reliability: Llama 3.3 70B or Qwen 2.5 72B if you have the VRAM. These approach cloud API quality across the board.

The pattern stays the same at every scale: define tools, run the model, validate the output, execute the function, feed results back. The only thing that changes is how many guardrails you need.

Related Guides

- Structured Output from Local LLMs
- llama.cpp vs Ollama vs vLLM
- Qwen Models Guide
- Mistral & Mixtral Guide
- Ollama vs LM Studio
- Local AI Planning Tool — VRAM Calculator

Sources: Ollama Tool Calling Docs, llama.cpp Function Calling, Docker Blog: Local LLM Tool Calling Evaluation, Llama 3.1 Prompt Format, NousResearch Hermes Function Calling, BFCL Leaderboard

Get notified when we publish new guides.

Subscribe — free, no spam

Source: https://insiderllm.com/guides/function-calling-local-llms/

Free guides for running AI locally