Best Local LLMs for Function Calling: Qwen 3.6, Gemma 4

February 11, 2026 · Updated: November 4, 2026

Download this guide as PDF

> **Quick Answer:** Function calling is what makes a chatbot into an agent — give the model a list of tools, it decides what to call and with which arguments. Local models can do this now: Ollama's tools API mirrors OpenAI's spec, llama.cpp has native handlers, and the gap to cloud APIs has mostly closed for single-tool tasks. This guide covers which models call tools reliably across VRAM tiers from 8GB through 24GB+, working code for both runtimes, an agentic-loop pattern with the production guards, and the failure modes that show up past the demo stage.

📚 **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 the Qwen 3.6 family ships with a dedicated `qwen3_coder` tool-call parser that closes most of the historical gap to cloud APIs on single-tool tasks. 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.

Updated 2026-11-04: the server command passes `-fa on`. Current llama.cpp's `-fa` / `--flash-attn` takes a value (`on`, `off` or `auto`) and no longer works bare, so the bare `-fa` swallowed the next argument. Originally said `llama-server --jinja -fa`.

Updated 2026-10-16: llama-server's default port moved from 8080 to 9931 in llama.cpp v0.7.0 (PR #30159). The server command now passes `--port 8080`, so the URLs below still work.

What's New (July 2026)

Function calling matured across the local-model lineup since this guide first shipped. The Qwen 2.5 era recommendations below still work — they just aren't the obvious picks anymore.

Qwen 3.6 family. Both the 27B dense and the 35B-A3B MoE variant ship with native tool calling via the `qwen3_coder` parser in vLLM/SGLang — and as of mid-2026 Ollama runs the text models directly too (`ollama pull qwen3.6:35b-a3b`), so the tools-API examples below work without standing up a separate serving stack. Community reports indicate better edge-case handling than the 2.5 line on nested JSON arguments, missing-parameter errors, and the choice not to call a tool, though dedicated BFCL-style benchmarks for 3.6 aren't yet publicly available. 35B-A3B runs on 16GB VRAM (`--cpu-moe` for the experts) and is the realistic default for agentic loops on a 3090 or 4090. See the Qwen Models Family Guide for setup.

Gemma 4 26B-A4B. Function calling works through the standard Gemma chat template, no special flags. One catch: Gemma 4 forces a reasoning trace by default, so tool-call outputs go to the `reasoning_content` field instead of `content` unless you pass `--jinja --chat-template-kwargs '{"enable_thinking":false}'`. I hit this on my head-to-head bench against Qwen 3.6. See the Gemma 4 guide for the chat-template story.

DeepSeek V4 Pro and Flash. Both ship with mature tool calling and MIT licenses. V4-Flash at 284B/13B active is realistic on serious homelabs; V4-Pro is workstation territory.

Active harnesses worth evaluating. PI Agent, OpenClaw, and Cline are all actively maintained options for production agent workflows as of mid-2026. Nous Research's Hermes-Function-Calling was an influential reference implementation for per-model tool-call formatting but hasn't seen updates since December 2025. See Best Local Alternatives to Claude Code for the full shortlist.

For new builds, start with Qwen 3.6 or Gemma 4. Qwen 2.5 7B / 14B still work and are now positioned as legacy budget picks in the model table below.

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 3.6-27B dense	27B	~17 GB	Strong (no published BFCL)	Top pick on 24GB cards; `qwen3_coder` parser
Qwen 3.6-35B-A3B	35B (3B active)	~24 GB	Strong (no published BFCL)	24GB clean / 16GB with `--cpu-moe`
Gemma 4 26B-A4B	26B (4B active)	~15 GB	Good (pass `enable_thinking=false`)	MoE alternative with broad runtime support
Qwen 3.5 9B	9B	~6 GB	Good	Recommended 8GB-tier pick
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.

Model	Size	VRAM (Q4)	Tool Accuracy	Best For
Mistral Small 24B	24B	~15 GB	Strong	Best agentic capabilities at this size.
Mixtral 8x7B	56B (13B active)	~24 GB	88% overall	Expert routing. Good multilingual.
Qwen 2.5 7B (legacy)	7B	~5-6 GB	Good (legacy)	Was the default pick pre-3.6
Qwen 2.5 14B (legacy)	14B	~9-10 GB	Good (legacy)	Solid budget pick before the 3.x line

The default recommendation: For 24GB cards, start with Qwen 3.6-35B-A3B (clean MoE) or Qwen 3.6-27B dense — both ship the `qwen3_coder` tool-call parser supported by vLLM and SGLang. For 16GB, run the 35B-A3B with `--cpu-moe`. For 8GB, Qwen 3.5 9B is the current general-purpose pick (Qwen 2.5 7B still works, but Qwen 3.5's hybrid attention edges it on long agent chains). The 0.933 F1 that Docker's June 2025 eval clocked at this tier belonged to Qwen 3 8B, not the 2.5 line — the full table is further down.

Llama 3.1 8B is the runner-up at the 8GB tier. 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-style XML tool format (`<tool_call>`, `<tool_response>`) is what Qwen 2.5 borrowed from Nous Research's Hermes line — vLLM and SGLang's automatic tool parsers handle it cleanly but it can confuse tools that expect OpenAI-style JSON. Qwen 3.6 supports both formats via its parser.

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": "qwen3.6:35b",
```

```
  "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

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="qwen3.6:35b",
    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 on \
  -hf unsloth/Qwen3.6-27B-GGUF:Q4_K_M \
  --port 8080
```

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

Tool Calling Request

```
curl http://localhost:8080/v1/chat/completions \
  -H "Content-Type: application/json" \
  -d '{
    "model": "qwen3.6",
    "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 = "qwen3.6:35b", 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

Per Docker's June 2025 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 3 14B came within 0.003 of GPT-4 in that eval. Docker hasn't published a follow-up covering Qwen 3.5 or 3.6, so the numbers above are the most recent published benchmark — read them as a snapshot of where local was in mid-2025, not where Qwen 3.6 sits today.

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	~80-150 (cloud frontier APIs)	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. Frontier-API 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.** Frontier models (GPT-5.6, Claude Opus 4.8) 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.

Extra Whitespace in `chat-template-kwargs` (Qwen 3.6)

Qwen 3.6 silently rejects template kwargs if you pass JSON with a space after the colon — `{"enable_thinking": false}` defaults to thinking mode instead of disabling it, and tool calls route to the reasoning channel. A widely-upvoted r/LocalLLaMA PSA flagged this across multiple setups.

Fix: Pass the JSON with no space around the colon: `'{"enable_thinking":false}'`. The parser is strict. Same rule applies to other Qwen-specific template kwargs.

Gemma 4 Tool Calls Routed to `reasoning_content`

If you switch to Gemma 4 26B-A4B and your tool-call output silently vanishes, the model is routing it to the `reasoning_content` field instead of `content` because Gemma 4 forces a reasoning trace by default. Your agent code reads `content` and gets nothing.

Fix: Launch llama-server with `--jinja --chat-template-kwargs '{"enable_thinking":false}'`. The load log will still print `thinking = 1` (cosmetic bug), but output correctly populates `content`. See my Gemma 4 head-to-head bench for the full context.

`num_ctx` VRAM Overflow Degrading Tool-Call Reliability

Setting `num_ctx` higher than what your VRAM actually fits causes silent CPU fallback that drops tool-call accuracy along with throughput — the model still emits structured calls, but format reliability degrades when half the layers are running on system RAM.

Fix: Match `num_ctx` to your hardware. On a 24GB 3090 with a Q4_K_M 27B model, ~16K-32K is the sweet spot; pushing 128K spills into system RAM. See num_ctx VRAM overflow for the diagnostic checklist.

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 qwen3.6:35b

# 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 3.5 9B on 8GB VRAM, or Qwen 2.5 7B if you're already running it.

For multi-step agents (chaining tools, branching logic): Qwen 3.6 35B-A3B on 24GB (or 16GB with `--cpu-moe`), or Qwen 3.6-27B dense on 24GB. Smaller models still lose coherence past 2-3 steps.

For maximum reliability: Qwen3-Coder-Next on 64GB+ unified memory, or DeepSeek V4-Flash via the DeepSeek API for frontier-adjacent capability without the local hardware. Llama 3.3 70B remains a capable dense alternative if you have 48GB+ VRAM.

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
- Best Local Models for PI Agent
- 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

A weekly email with every new guide and measured benchmark.

Subscribe — free, no spam

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

Free guides for running AI locally