Building AI Agents with Local LLMs: A Practical Guide

February 23, 2026 · by Mark Bartlett

Download this guide as PDF

> **Quick Answer:** Local AI agents need models that can reliably follow multi-step instructions and call tools. The minimum viable model is Qwen 3 8B for simple single-tool tasks, but for anything multi-step you want Qwen 3 32B or Mistral Small 24B — and 70B-class models for complex reasoning chains. Small models (1-7B) mostly fail at agentic tasks. Budget 4-8GB of extra VRAM beyond the model itself for the long context windows agents require.

📚 **More on this topic:** How OpenClaw Works · Function Calling with Local LLMs · Best Models for OpenClaw · Planning Tool

AI agents are the most hyped concept in the LLM space right now. Most of the hype targets cloud APIs (GPT-4o, Claude, Gemini). But if you're reading InsiderLLM, you want to know: can you build agents that run entirely on your own hardware?

The honest answer: yes, with caveats. The model matters enormously. The framework matters less than you think. And security matters more than anyone wants to admit.

This guide covers the architecture, the models that actually work, the frameworks worth considering, and includes a complete working agent in ~50 lines of Python.

What is an AI agent?

An agent is an LLM that takes actions. It doesn't just generate text. It actually does things.

A chatbot generates a response. An agent reads your question, decides it needs to check something, calls a tool to check it, reads the result, decides what to do next, calls another tool, and eventually gives you an answer based on real actions it took.

Concretely, agents can browse the web (fetching URLs, scraping data, running searches), execute code (Python, shell commands, SQL), manage files, call APIs, and control other programs like git or docker.

The main difference from a chatbot: **the LLM is in a loop**. It observes, thinks, acts, then observes the result and repeats. This loop continues until the task is done or the model decides it's finished.

You've already seen this if you've used Claude Code, GitHub Copilot, or OpenClaw. These are all agents, LLMs wrapped in tool-calling loops.

The ReAct loop

Almost every agent framework uses some variant of the ReAct (Reasoning + Acting) pattern. Here's the core loop, stripped of all framework overhead:

```
1. OBSERVE  → The model receives the current state (user query + tool results so far)
2. THINK    → The model reasons about what to do next
3. ACT      → The model calls a tool (or decides it's done)
4. REPEAT   → The tool result goes back into context, return to step 1
```

That's it. Every agent framework — LangChain, CrewAI, AutoGen, smolagents — is fundamentally a wrapper around this loop plus tool definitions.

Tool calling

The model needs to output structured tool calls. There are two approaches:

With **native tool calling**, models like Qwen 3, Llama 3.3, and Mistral support function calling as part of their chat template. Ollama exposes this via the `tools` parameter. The model outputs a structured JSON object specifying which function to call and with what arguments.

With **text-based tool calling**, the model outputs something like `<tool_call>{"name": "read_file", "args": {"path": "/tmp/data.txt"}}</tool_call>` in its response, and your code parses it out. Less reliable, but works with any model.

Native tool calling is significantly more reliable. Use it when available.

Memory

Agents have two memory problems.

Short-term memory is the context window. Everything the agent has seen in the current session lives there. Each tool call and result adds tokens. A 10-step agent interaction can easily

consume 8K-16K tokens. With long context, you can sustain longer sessions, but you pay in VRAM for KV cache.

Long-term memory means persisting information across sessions. Most frameworks handle this with vector databases or simple file storage. For local setups, ChromaDB or just writing to disk works fine. This is less of a model problem and more of an engineering problem.

Which models actually work for agents?

This is where most guides lie to you. They show demos with GPT-4 and imply your local 7B model can do the same thing. It can't.

Agentic tasks are **the hardest thing you can ask an LLM to do**. The model needs to:

1. Understand the task
2. Plan multiple steps
3. Output perfectly structured tool calls (one bad JSON = broken agent)
4. Interpret tool results correctly
5. Know when to stop

Here's the honest breakdown:

Model tiers for agent tasks

Tier	Models	Agent Capability	VRAM (Q4)
Fails	Most 1-4B models	Cannot reliably produce structured tool calls. Don't bother.	2-4 GB
Barely works	Qwen 3 8B, Llama 3.1 8B	Single-tool calls on simple tasks. Falls apart on multi-step.	6-8 GB
Minimum viable	Qwen 3 14B, Gemma 3 12B	Can handle 2-3 step chains with clear tool definitions. Occasional hallucinated calls.	10-12 GB
Sweet spot	Qwen 3 32B, Mistral Small 24B	Reliable multi-step tool use. Good at knowing when to stop. Handles 5-10 step chains.	16-20 GB
Best local	Llama 3.3 70B, Qwen 3 72B, DeepSeek-R1 70B	Complex reasoning chains, error recovery, multi-tool coordination. Approaches cloud API quality.	36-42 GB

The small model problem

Let me be direct: **models under 8B parameters mostly fail at agentic tasks.**

They fail because structured output is fragile at that scale. A 3B model will output almost-valid JSON, and almost-valid isn't valid. One missing quote breaks your entire agent loop. Multi-step planning is weak too; small models can handle "call this one function" but fall apart at "first check X, then based on the result, do Y or Z." They also hallucinate tool calls constantly. A 7B model will cheerfully call functions that don't exist, pass wrong argument types, or invent parameters. And error recovery is nonexistent. When a tool returns an error, small models either repeat the same call or spiral into nonsense.

I've tested this extensively. A Qwen 3 4B model hooked up to a file management agent will:

- Call `read_file` on paths that don't exist, then call it again with the same path
- Output `write_file` with the content argument missing
- Decide it's "done" after one step when the task clearly needs three
- Produce tool call JSON with trailing commas, unescaped quotes, or missing brackets

If you have 24GB of VRAM, use Qwen 3 32B. If you have 12-16GB, use Qwen 3 14B and keep tasks simple. If you only have 8GB, you can try Qwen 3 8B for basic single-tool workflows, but set your expectations low.

VRAM requirements for agent workloads

Agents need more VRAM than basic chat because of context length. A chatbot might work fine at 2K-4K context. An agent routinely needs 8K-32K tokens as tool calls and results accumulate.

Longer context = larger KV cache = more VRAM. Here's what to budget:

Agent VRAM table

Model (Q4_K_M)	Base VRAM	+8K ctx	+16K ctx	+32K ctx	+64K ctx
Qwen 3 8B	5.5 GB	6.5 GB	7.5 GB	9.5 GB	13.5 GB
Qwen 3 14B	9.5 GB	10.5 GB	12 GB	14.5 GB	19.5 GB
Mistral Small 24B	14 GB	15.5 GB	17 GB	20 GB	26 GB
Qwen 3 32B	19 GB	20.5 GB	22 GB	25 GB	31 GB

Model (Q4_K_M)	Base VRAM	+8K ctx	+16K ctx	+32K ctx	+64K ctx
Llama 3.3 70B	40 GB	42 GB	44 GB	48 GB	56 GB

Rule of thumb: Budget your model's base VRAM + 50% for agent context headroom. If your model needs 19GB at base, plan for 28-30GB of available VRAM.

This is why agents on consumer GPUs are hard. A Qwen 3 32B model doing agent work at 32K context needs ~25GB — that's your entire RTX 3090/4090 budget with almost nothing left for the OS. Check VRAM requirements for exact numbers per model.

Framework comparison

You don't need a framework to build agents. But frameworks save time on tool definitions, output parsing, and memory management. Here's what's out there:

Framework	Language	Complexity	Best For	Local LLM Support	Overhead
LangChain/ LangGraph	Python	High	Complex multi-agent workflows	Good (Ollama, llama.cpp)	Heavy, massive dependency tree
CrewAI	Python	Medium	Role-based multi-agent teams	Good (Ollama)	Medium
AutoGen (Microsoft)	Python	Medium-High	Conversational multi-agent	Good	Medium
smolagents (HuggingFace)	Python	Low	Simple single-agent tasks	Good	Light, minimal deps
OpenClaw	TypeScript	Medium	Coding agents, general tasks	Excellent (built for local)	Medium
Roll your own	Any	Lowest	Learning, custom needs	You decide	Zero

My recommendations

If you're learning, roll your own. The loop is ~50 lines. You'll understand agents better in an afternoon than a week of LangChain docs.

For simple tasks, I like smolagents from HuggingFace. Minimal API, clean abstractions, works well with Ollama.

For coding agents, go with OpenClaw and a good local model. It's purpose-built for code and has solid Ollama integration.

For production multi-agent systems, use LangGraph if you need flexibility, or CrewAI if you want something more opinionated. Both work, both have warts.

I'd skip vanilla LangChain (the non-graph version). It adds complexity without proportional value for most local setups. AutoGen is powerful but its API changes frequently.

OpenClaw + Ollama integration

OpenClaw is the most popular open-source coding agent. It connects to local models through Ollama's OpenAI-compatible API:

```
# Start Ollama with your agent model
ollama run qwen3:32b

# Configure OpenClaw to use local endpoint
# In OpenClaw settings:
# API Base: http://localhost:11434/v1
# Model: qwen3:32b
```

OpenClaw works best with models that score well on function calling benchmarks. For model recommendations, see our best models for OpenClaw guide.

The honest take: OpenClaw with a local 32B model is usable for straightforward coding tasks. For complex multi-file refactors, you'll hit the reasoning ceiling that cloud models don't have. Review the security guide before giving any agent access to your codebase.

Security: agents are dangerous

This section isn't optional. Read it.

An agent with code execution capabilities can **delete files, install malware, exfiltrate data, and brick your system**. This isn't theoretical. It happens when:

- The model hallucinates a shell command (`rm -rf /` is a popular hallucination)
- A prompt injection in fetched web content hijacks the agent
- The model misunderstands "clean up old files" as "delete everything"

Mandatory safety rules

1. **Sandbox everything.** Run agents in Docker containers, VMs, or at minimum a restricted user account. Never run an agent as root.
2. **Allowlist tools.** Don't give agents `execute_shell`. Give them specific, scoped tools like `list_files(directory)` that can only operate in designated paths.
3. **Validate all tool arguments.** Check paths are within allowed directories. Sanitize inputs. Reject anything that looks like path traversal.
4. **Human-in-the-loop for destructive actions.** Any tool that writes, deletes, or modifies should require confirmation for anything outside a sandbox directory.
5. **Kill switch.** Set a maximum number of iterations (10-20 for most tasks). An agent stuck in a loop will burn through your context window and potentially execute the same dangerous action repeatedly.

```
# MINIMUM viable path safety
import os

ALLOWED_DIR = "/tmp/agent-sandbox"

def safe_path(requested_path: str) -> str:
    """Resolve path and verify it's within the sandbox."""
    resolved = os.path.realpath(os.path.join(ALLOWED_DIR, requested_path))
    if not resolved.startswith(os.path.realpath(ALLOWED_DIR)):
        raise ValueError(f"Path {requested_path} escapes sandbox")
    return resolved
```

For more on securing local agents, see the OpenClaw security guide.

Practical example: file management agent

Here's a complete, working agent in ~50 lines of Python. It uses the Ollama Python library and can list, read, and write files in a sandboxed directory.

Prerequisites

```
pip install ollama
ollama pull qwen3:14b   # minimum viable; qwen3:32b recommended
mkdir -p /tmp/agent-sandbox
```

The complete agent

```
import json
import os
import ollama

SANDBOX = "/tmp/agent-sandbox"
MODEL = "qwen3:14b"
MAX_ITERATIONS = 10

# --- Tool definitions (Ollama format) ---
tools = [
    {
        "type": "function",
        "function": {
            "name": "list_files",
            "description": "List files in a directory relative to the sandbox",
            "parameters": {
                "type": "object",
                "properties": {
                    "path": {"type": "string", "description": "Relative directory path (default
                },
            },
        },
    },
    {
        "type": "function",
        "function": {
            "name": "read_file",
            "description": "Read the contents of a file relative to the sandbox",
            "parameters": {
```

```
                "type": "object",
                "properties": {
                    "path": {"type": "string", "description": "Relative file path"}
                },
                "required": ["path"],
            },
        },
    },
    {
        "type": "function",
        "function": {
            "name": "write_file",
            "description": "Write content to a file relative to the sandbox",
            "parameters": {
                "type": "object",
                "properties": {
                    "path": {"type": "string", "description": "Relative file path"},
                    "content": {"type": "string", "description": "Content to write"},
                },
                "required": ["path", "content"],
            },
        },
    },
]

# --- Tool implementations (sandboxed) ---
def safe_resolve(rel_path: str) -> str:
    resolved = os.path.realpath(os.path.join(SANDBOX, rel_path))
    if not resolved.startswith(os.path.realpath(SANDBOX)):
        raise ValueError(f"Path escapes sandbox: {rel_path}")
    return resolved

def list_files(path: str = ".") -> str:
    target = safe_resolve(path)
    if not os.path.isdir(target):
        return f"Error: {path} is not a directory"
    return "\n".join(os.listdir(target)) or "(empty directory)"

def read_file(path: str) -> str:
    target = safe_resolve(path)
    if not os.path.isfile(target):
        return f"Error: {path} not found"
    with open(target, "r") as f:
        return f.read()[:4000]  # cap at 4K chars

def write_file(path: str, content: str) -> str:
    target = safe_resolve(path)
    os.makedirs(os.path.dirname(target), exist_ok=True)
    with open(target, "w") as f:
```

```
        f.write(content)
    return f"Wrote {len(content)} chars to {path}"

TOOL_MAP = {"list_files": list_files, "read_file": read_file, "write_file": write_file}

# --- ReAct loop ---
def run_agent(user_task: str):
    messages = [
        {"role": "system", "content": (
            "You are a file management assistant. Use the provided tools to complete tasks. "
            f"All file paths are relative to the sandbox directory. "
            "Call tools as needed. When the task is complete, respond without tool calls."
        )},
        {"role": "user", "content": user_task},
    ]

    for i in range(MAX_ITERATIONS):
        response = ollama.chat(model=MODEL, messages=messages, tools=tools)
        msg = response["message"]
        messages.append(msg)

        # No tool calls = agent is done
        if not msg.get("tool_calls"):
            print(f"\n✅ Agent done ({i+1} iterations):\n{msg['content']}")
            return msg["content"]

        # Execute each tool call
        for tc in msg["tool_calls"]:
            func_name = tc["function"]["name"]
            args = tc["function"]["arguments"]
            print(f"  🔧 {func_name}({json.dumps(args)})")

            try:
                result = TOOL_MAP[func_name](**args)
            except Exception as e:
                result = f"Error: {e}"

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

    print("⚠️  Max iterations reached")
    return None

if __name__ == "__main__":
    run_agent("List all files, then create a file called summary.txt with a list of what you fou
```

What this does

1. Defines three tools with JSON schemas that Ollama understands
2. Implements each tool with path sandboxing
3. Runs a ReAct loop: send messages to the model, execute any tool calls, feed results back
4. Stops when the model responds without tool calls (task complete) or hits the iteration limit

Running it

```
# Create some test files first
echo "Hello world" > /tmp/agent-sandbox/hello.txt
echo "Some data" > /tmp/agent-sandbox/data.csv

python agent.py
```

Expected output with Qwen 3 14B:

```
  🔧 list_files({})
  🔧 read_file({"path": "hello.txt"})
  🔧 read_file({"path": "data.csv"})
  🔧 write_file({"path": "summary.txt", "content": "Files found:\n- hello.txt: Contains 'Hello wo

✅ Agent done (3 iterations):
I found 2 files in the sandbox and created summary.txt with a description of each.
```

With Qwen 3 8B, expect occasional failures: malformed JSON in tool calls, skipping the `list_files` step and going straight to writing, or calling `read_file` without arguments. With a 32B model, it handles this task perfectly every time in my testing.

Tips for reliable local agents

Keep tool definitions simple. Fewer parameters, clear descriptions. The model reads these descriptions, and vague descriptions produce vague tool calls.

Use native tool calling. Don't parse free-text tool calls unless you have to. Ollama's tool calling support with Qwen 3 and Mistral models is solid enough.

Start with 2-3 tools max. More tools means more choices for the model to screw up. Add complexity after the basics work.

Set temperature to 0 for agents. You want deterministic, reliable tool calls, not creative ones. `temperature=0` in your Ollama call reduces hallucinated arguments.

Log everything. Print every tool call and result. When (not if) something goes wrong, you need the full trace.

Test with known tasks first. Before pointing an agent at real work, run it through 10 scripted tasks and check every tool call. Measure the failure rate. If it's above 10%, your model is too small.

Think about context length. Each iteration adds ~500-2000 tokens of tool calls and results. At 10 iterations, that's 5K-20K tokens of accumulated context. Make sure your model and VRAM can handle it.

What's next for local agents

The gap between cloud and local agents is closing fast. A year ago, only GPT-4-class models could handle multi-step tool use. Today, Qwen 3 32B running on a single RTX 3090 handles 80% of what you'd throw at Claude or GPT-4o.

What's still missing locally:

- Very long reasoning chains (20+ steps). 70B models can do this but need dual GPU setups.
- Multi-agent coordination. Running two 32B models simultaneously needs 48GB+ VRAM.
- Real-time web agents. Possible but slow; the model inference latency makes interactive browsing painful.

The practical sweet spot right now: single-agent, 5-10 step tasks, with a 14B-32B model, on 24GB of VRAM. That covers file management, code generation, data processing, and simple API integrations. For anything more complex, either scale up to 70B with more VRAM or accept that a cloud API call might be the pragmatic choice.

Related guides

- Function Calling with Local LLMs: deep dive on native tool calling, JSON mode, and which models support it
- How OpenClaw Works: architecture of the most popular open-source coding agent
- Best Local Models for OpenClaw: benchmarked recommendations for agent-capable models

- Structured Output with Local LLMs: JSON schemas, constrained generation, and grammar enforcement

---

Source: https://insiderllm.com/guides/local-ai-agents-guide/