ChatGPT

Building a Python Tool-Calling Loop for Local LLMs

This guide explains how to extend a Python-based agent loop to enable **function calling** (tools) with a local OpenAI-compatible LLM endpoint (e.g. LM Studio + Qwen-3). We cover how to register JSON-schema tools, implement streaming & asyncio for concurrency, harden tool execution (sandboxing, allow-lists, cross-OS considerations), manage configuration via environment settings, and handle common pitfalls like token limits, recursive loops, and timeouts. The examples favor open-source libraries and Python-first patterns over closed frameworks.

Defining Tools with JSON Schema

Large Language Models can be given **tool definitions** (functions) described by a name, description, and a JSON Schema for parameters. This schema tells the model what arguments the function expects, enabling the model to output a structured request for that function. For example, here we define a simple weather API tool with required `location` and an optional `unit` parameter:

```
# Define available function tools with JSON-schema parameters
functions = [
    {
        "name": "get_current_weather",
        "description": "Get the current weather in a given location",
        "parameters": {
            "type": "object",
            "properties": {
                "location": {
                    "type": "string",
                    "description": "The city and state, e.g. San Francisco, CA",
                    "maxLength": 20
                },
                "unit": {
                    "type": "string",
                    "enum": ["celsius", "fahrenheit"]
                }
            },
            "required": ["location"]
        }
    }
]
```【44†L300-L308】【44†L312-L319】

Such a definition can be passed in a chat request (e.g. via the OpenAI `/v1/chat/completions` API) under a `tools` or `functions` parameter. Many

frameworks can auto-generate JSON schemas from Python function signatures. For instance, Microsoft's **AutoGen** library wraps a Python function into a `FunctionTool` and automatically produces a schema (name, description, parameter types) for the model to follow【11†L199-L207】. The LLM uses these schemas to decide when and how to call a function. In practice, you might register tools like `"execute_python"` (to run Python code), `"git_status"` (to check Git status), or file editors in the same way – by providing a name, description, and parameter schema describing any arguments (e.g. file path, content, etc.).

Example – Function Call Schema: Below is an example JSON schema auto-generated for a stock price lookup tool. It shows the structure the LLM will conform to when calling the function (including required parameters and types):

```json
{
  "name": "get_stock_price",
  "description": "Get the stock price.",
  "parameters": {
    "type": "object",
    "properties": {
      "ticker": {"type": "string", "description": "ticker", "title": "Ticker"},
      "date":   {"type": "string", "description": "Date in YYYY/MM/DD", "title": "Date"}
    },
    "required": ["ticker", "date"],
    "additionalProperties": false
  }
}
```【11†L203-L212】【11†L213-L221】

Why this matters: Defining tools with a JSON schema ensures the LLM's tool-call requests are well-structured. The model will output a JSON snippet matching the schema when it decides to invoke a tool, which your code can then safely parse and route to the appropriate function.

Implementing the Tool-Calling Loop

Once tools are defined and provided to the model, the agent loop is responsible for **detecting tool call requests, executing the tools, and feeding results back** to the LLM. Typically the loop looks like:

1. **Send user prompt to LLM** along with the list of tool schemas.
2. **Check LLM response** – if it contains a function/tool call, parse out the `name` and `arguments`.
3. **Execute the tool** (your Python function or API) with those arguments.
4. **Insert the tool's result** into the conversation history, and prompt the LLM again (this time without tools enabled) so it can use the result to continue

````
its answer.
5. **Repeat** if the LLM makes further tool requests; otherwise, return the
final answer.

In code, this loop can be implemented as follows (pseudocode adapted from LM
Studio docs):

```python
response = client.chat_completions.create(..., tools=tools)  # initial LLM call
if response.has_tool_calls:
    for tool_call in response.tool_calls:
        func_name = tool_call.name       # e.g. "get_delivery_date"
        args = tool_call.arguments       # e.g. {"order_id": "123"}
        result = execute_function(func_name, args)  # call the actual tool
        # Append the tool call and its result to the message history
        messages += [
            {"role": "assistant", "function_call": {"name": func_name,
"arguments": args}},
            {"role": "tool", "content": format_result(result)}
        ]
    # Call LLM again with updated messages (no tools this time) to get final
answer
    final_response = client.chat_completions.create(..., messages=messages,
tools=None)
else:
    final_response = response  # no tool used, this is final answer
````

In this flow, the assistant's tool request and the tool's output are added to the chat history so the model can incorporate them in the next turn [4] [5]. Notably, after using a tool, we call the model **without the tool list** to encourage it to produce a final user-facing answer instead of looping into another tool call [6]. This prevents recursive or endless tool use.

Why this matters: This "ReAct" style loop ensures the LLM and tools work in tandem. The LLM can ask for computations or data it cannot produce on its own, and your code mediates these requests and responses step by step [7] [8]. The separation of concerns (LLM decides *what* to do; Python executes *how* to do it) makes the system extensible and debuggable.

Streaming Responses and Async Tool Execution

Streaming LLM output: For better responsiveness, you can stream the LLM's response tokens as they are generated. When using the OpenAI-compatible API, set `stream=True` on the chat completion call, and iterate over the resulting generator. For example, using a local OpenAI-like client (such as LiteLLM):

```
python
response = completion(model="openai/gpt-4o", messages=messages, stream=True)
for part in response:
```
```

```
    # Each part is a ChatCompletionChunk; print incremental content
    print(part.choices[0].delta.content or "", end="") [9]
```

When streaming with function calls, note that **function call data also streams in pieces.** The model will send the function `name` and `arguments` gradually in the `delta.tool_calls` fields. For example, a call like `get_current_weather(location="San Francisco")` might arrive as several chunks: one with `"function.name": "get_current_weather"`, and subsequent chunks with partial JSON of the arguments (e.g. `"arguments": "{"`, then `"arguments": "\"location\": \"San"`, etc.) [10] [11]. Your loop should accumulate these chunks to reconstruct the full tool request before execution [12]. In practice, the OpenAI SDK handles this for function calls – you'll get a final `function_call` object once complete – but if using lower-level streaming, be mindful of assembling the JSON.

Async tool execution: To maximize throughput, especially if a model call can request multiple tools at once or if tools are I/O-bound, you should incorporate `asyncio`. Python's `asyncio.create_task` or `asyncio.gather` allows running multiple tool functions in parallel. For example, if an LLM outputs a list of tool calls `actions` to perform, you can do:

```python

# Concurrently execute each tool action (async functions assumed)

results = await asyncio.gather(*(execute_tool_action(action) for action in actions)) for result in results: yield result # or handle result ``` [13]

This snippet uses `asyncio.gather` to schedule all tool executions concurrently, then yields each result as it completes [13] [14]. By handling tools in parallel, an agent can significantly reduce latency when multiple independent steps are needed. For example, an LLM might ask to fetch data from two APIs before responding – those can be done in parallel if your tool functions are `async`.

**Back-pressure & resource management:** When using concurrency, implement limits to avoid overload. For instance, use an `asyncio.Semaphore` to cap the number of parallel tasks (preventing 100 tools from running at once). The sandbox server **AgentExecMPC** imposes a max of 4 concurrent processes by design [15] [16]. Streaming responses also require managing back-pressure – e.g. reading the model's token stream and not flooding it with new input until it's ready. Libraries like `asyncio.Queue` can help buffer streaming outputs or tool results to be consumed by other coroutines in a controlled way.

*Why this matters:* Streaming improves user experience with faster partial answers, and `asyncio` concurrency can dramatically speed up multi-tool workflows. Combined, streaming + async let your agent feel real-time and efficient. Just take care to synchronize the assembly of streamed function-call parts, and to guard against race conditions or resource exhaustion when running many tasks in parallel.
```

Secure and Hardened Tool Execution

Allowing an LLM to execute code or shell commands is powerful but **dangerous**. You must harden this execution environment to protect the host system. Key strategies include:

- **Sandboxing the runtime:** Run code in a restricted environment (Docker container, VM, or sandbox library). For example, *AutoGen* runs its `PythonCodeExecutionTool` inside a Docker container to isolate file system and processes [17]. In code, it looks like:

```
python
code_executor = DockerCommandLineCodeExecutor()
await code_executor.start()                  # Launch sandboxed Docker container
code_tool = PythonCodeExecutionTool(code_executor)
result = await code_tool.run_json({"code": "print('Hello')"})   # executes
inside container
```
[18] [19]

This ensures any Python code from the LLM runs in a throwaway container with limited OS permissions. On Linux, tools like **seccomp** (secure computing mode) or **chroot jails** can also sandbox a process to restrict syscalls or filesystem access, respectively. For instance, you could launch shell commands via a wrapper that applies a seccomp filter (disallowing dangerous syscalls) or run them in a `chroot` to confine accessible directories [20]. On Windows, containers or virtualization (e.g. Windows Sandbox or WSL with Docker) may be used since seccomp/chroot are Linux-specific.

- **Allow/Deny-lists for commands:** Restrict what shell commands or modules the agent can use. **Auto-GPT's CodeExecutor** component, for example, has an `execute_local_commands` flag (default False) and supports an *allowlist* or *denylist* of shell commands in its config [20]. By default it uses an empty allowlist, meaning no shell commands are permitted unless explicitly added, and it warns that without further sandboxing this is not secure for production [20] [21]. You can maintain a list of safe commands (e.g. `["ls","grep","git status"]`) and filter any LLM-requested command against it before execution. Regex can be used to validate command format (for example, forbid special characters like `&&` or `;` that chain commands). Start with a very restrictive set and expand only as needed ("principle of least privilege").

- **Language-level sandboxing:** If executing generated code, consider running it with a restricted interpreter. The **HuggingFace *smolagents*** project provides a `LocalPythonExecutor` that interprets code ASTs with heavy restrictions: by default *no imports* unless explicitly allowed, *no attribute access* to disallowed modules, a cap on the number of operations to prevent infinite loops, and errors for any operation not on an approved list [22] [23]. For example, it will refuse to import `os` or run shell escapes, unless you explicitly permit them (and even then, will catch attempts to call dangerous functions) [24] [25]. This kind of "secure interpreter" is Python-specific but very useful for safe evaluation of code without full OS sandbox overhead.

- **Resource limits & timeouts:** Always impose time and memory limits on tool execution. If a tool hangs or goes rogue (intentional or not), your agent should terminate it. For shell processes, use subprocess timeouts or run them via an orchestration tool that kills after N seconds. In the sandbox server example above, each command has a configurable timeout (60s by default, 300s max) [15].

Similarly, limit memory/CPU if possible (cgroups in Docker or Linux), and file I/O (e.g. restrict to a temp directory).

- **Non-root execution & file access limits:** Run tools as a non-privileged user. The AgentExecMPC container, for instance, uses a dedicated `agent` user (UID 10001) and confines all operations to a `/workspace` directory [26]. In your own setup, you might create a throwaway user account with no sudo rights for the agent. Also limit filesystem access: **AutoGPT** agents only read/write files in their designated `workspace` folder by default, and will refuse paths outside it unless an override flag is set [27]. (It explicitly advises against lifting this restriction except if running in a fully isolated sandbox [28].) Designing your file-editing tools to only affect a specific directory (and perhaps with filename whitelists or size limits) can prevent the agent from tampering with arbitrary files on your disk.

- **Emergency stop & monitoring:** Have a way to interrupt or shut down the agent if it goes out of bounds. This could be a manual kill switch or an automated detector if the agent starts looping unexpectedly or producing dangerous requests. Also log all commands run and their outputs for audit—this helps in debugging and in spotting malicious patterns.

Why this matters: Without proper hardening, an AI agent with tool access can accidentally or maliciously run destructive commands (`rm -rf /` is the classic nightmare). By sandboxing execution, filtering allowed actions, and running with least privilege, you greatly reduce risk. Many open-source agent frameworks implement some of these measures (e.g. non-root Docker containers, allowlists, custom interpreters) to make tool use **"safe by default"** [29] [30]. Always assume an advanced prompt injection might trick the model—so build multiple safety layers.

Configuration and Environment Management

A flexible agent should allow configuration of endpoints, models, API keys, and tool settings **without code changes**. Good practices include:

- **.env Files for Secrets and Settings:** Use a `.env` file or environment variables to store API keys, model names, etc., rather than hard-coding them. For example, you might have: `bash OPENAI_API_KEY=<your-api-key> OPENAI_API_BASE=http://localhost:1234/v1 MODEL_NAME=lmstudio-community/qwen2.5-7b-instruct RESTRICT_TO_WORKSPACE=True DISABLED_COMMANDS=execute_python_code,execute_python_file` Your code can load these via a library like `python-dotenv` or Pydantic's BaseSettings [31]. Auto-GPT, for instance, provides a `.env.template` listing dozens of config options. At runtime it reads those into a config, and warns that env vars are overridden by explicit config files if provided [32]. In Auto-GPT's classic mode, `RESTRICT_TO_WORKSPACE` and `DISABLED_COMMANDS` (as shown above) are two env toggles used to secure the agent's behavior [27] [33].

- **Typed Config Objects:** Consider defining a Pydantic model or dataclass for settings (e.g. `Settings(model_name: str, use_stream: bool, max_iterations: int, ...)`), which can load from env vars. This gives you **type-checked** configuration and easy defaults. Pydantic's settings management makes it easy to load nested configurations from env with prefixes, etc. [31].

- **Runtime model/endpoint selection:** Your loop should allow choosing a different model or backend at startup. For example, if using OpenAI's Python SDK pointed at LM Studio, you can set the base URL to switch between local and cloud: `python`
  ```
  import openai
  openai.api_base = os.getenv("OPENAI_API_BASE", "http://localhost:1234/v1")
  openai.api_key = os.getenv("OPENAI_API_KEY", "dummy-key")
  response = openai.ChatCompletion.create(model="gpt-4o-mini", messages=[...])
  ```
  In the LM Studio example below, the `OpenAI` client is configured to hit the local server and a specific model ID is passed in the request, which could come from an env var or CLI argument:
  `python`
  ```
  client = OpenAI(base_url="http://localhost:1234/v1", api_key="lm-studio")
  completion = client.chat.completions.create(
      model="model-identifier",
      messages=[ ... ]
  )
  ```
  [34]

This approach lets you swap models (e.g. a smaller Qwen for testing vs a larger one for production) or even switch out the LLM provider (perhaps targeting an OpenAI cloud model vs. a local one) by changing an environment setting. Similarly, tool configurations like file paths or API endpoints could be configurable in a settings file.

- **Logging and debug config:** It's helpful to make verbose logging toggle-able via env (e.g. `DEBUG=1`). During development, you can log full LLM prompts and tool outputs. In production, you might disable those or log to a file. Many agent frameworks provide debug modes; for example, you can run Auto-GPT with `--debug` to get detailed logs [35].

Why this matters: Clean config management makes your agent **portable and adjustable**. You might deploy it on a server with different environment variables, or open-source it and let others provide their own keys and model choices. By not hard-coding values, you also reduce the chance of accidentally leaking secrets. Using typed config classes helps catch misconfigurations early (e.g. if a required env var is missing or of the wrong form).

Handling Token Limits, Loops, and Timeouts

Finally, when running an autonomous loop, be aware of common failure modes and how to mitigate them:

- **Token Limit Management:** Each model has a context length (e.g. 4k, 8k tokens) beyond which it cannot process prompts. Long conversations or verbose tool results can approach this limit, causing model errors or truncated outputs. To avoid this, implement a **message buffer**: keep a rolling window of recent messages and prune older ones. A typical strategy is to only send the last N messages or last M tokens of history to the model ("Rolling Context") [36]. Another strategy is to **summarize old interactions**: when the conversation grows too long, use the model (or a separate summarizer) to compress earlier exchanges into a shorter synopsis, and include that instead of the full text [36]. This "Summarized Memory" approach retains important info while freeing up token space. A third strategy is **topic-based condensation**: if the conversation has distinct topics, you can retain detailed context only for the relevant topic at hand, and summarize or drop others [36].

Also consider using the model's token usage info to monitor consumption. Many APIs return usage metrics; for example, OpenAI returns `usage.total_tokens`. You can set a threshold (say 90% of max context) to trigger a trimming of history. If a tool returns a very large result (like reading a big file), you might post-process that (e.g. summarize the file) before appending it to chat history, to save tokens.

- **Preventing Infinite Loops/Recursion:** An agent might get stuck in a loop of calling the same tool repeatedly or bouncing between the LLM and tool without making progress. To guard against this, implement a **max iterations** or recursion limit. For instance, in LangChain's agent executor you can set `max_iterations=3` so it will error out after 3 tool uses without finishing [37]. The LangChain docs note: *"To control agent execution and avoid infinite loops, set a recursion limit (max number of steps)"* [38]. In your own loop, you could simply count how many cycles have run and break if it exceeds a safe limit (possibly returning an apologetic answer or requiring user intervention). Additionally, after each tool + model cycle, you might check if the conversation is repeating itself — if the last tool call is the same as a previous one with no new information, that's a sign to stop.

Another vector is **self-reflection prompts**: some agent frameworks have the LLM reflect if it's not making progress. But a simpler and effective measure is a hard cap on steps and clear logging when it's reached (so you can analyze the cause).

- **Tool Timeout & Error Handling:** We touched on timeouts earlier, but it's worth emphasizing: any external call (code execution, web request, etc.) should have a reasonable timeout. If a tool times out or fails, decide how the agent should respond. You might catch the exception and feed a special "error result" back to the LLM (e.g., content: `"Tool X failed with timeout"`). Modern function-calling allows you to return function errors as assistant messages too. For example, you might return a placeholder result like `"ERROR: Tool timed out"` as the tool's content – a robust model can decide to apologize or try an alternative approach. Without handling timeouts, your whole agent might hang. Use Python's `asyncio.wait_for` or subprocess `timeout` parameter, or an external watchdog. The sandbox service mentioned implements automatic process cleanup after the timeout hits [39].

- **Memory Leaks and Resource Cleanup:** If your agent writes files or allocates memory (e.g. in tools), clean up if those are no longer needed. Delete temporary files, close database connections, etc. to avoid crashes over long runs. Agents that autonomously code or install packages (like some AutoGPT plugins) should ideally do so in an isolated environment that can be reset periodically to avoid clutter.

- **User Handoff on Failure:** If the agent cannot complete a task due to tool unavailability, hitting token limit, or other errors, it's better to surface that to the user than to get stuck. For example, if all else fails, the agent might respond: *"I'm sorry, I'm running out of context and can't continue. You may need to start a new session."* This is preferable to incoherent output when over limit [40]. Similarly, catch exceptions in the main loop and present a controlled failure message instead of letting the program crash.

Why this matters: Robust handling of these issues turns a toy agent into a production-ready one. Token limits are a fundamental constraint; planning around them (with pruning or summarizing) is crucial for long conversations. Preventing infinite loops keeps the agent from getting stuck or racking up API bills unnecessarily. And graceful error and timeout handling ensures the system doesn't just freeze if something

goes wrong – it either corrects course or fails safely. As a developer, you'll want to log these events (e.g. "Hit token limit, summarized memory" or "Max iterations reached, aborting") so you can continuously improve the agent's prompts or toolset to handle such cases more elegantly in the future.

Useful Resources (with Examples)

Below is a quick reference list of the resources used, each demonstrating key aspects of the above guide:

- **LM Studio Tool Use Docs** – Illustrates the full tool-calling loop with Qwen, including how tools are defined in JSON and how the conversation is updated with tool results [4] [6]. Useful for understanding the overall agent flow with function calls.
- **Qwen Function Calling (ReadTheDocs)** – Alibaba's Qwen models support function calling. The docs show how to construct the `tools` list with JSON schemas and provide Hermes-style prompt templates for tool use [41] [42]. Helpful for seeing a real JSON schema in context and the prompt format required by certain models.
- **Local LLM Function Calling (GitHub rizerphe)** – Open-source library enforcing JSON schema on local HuggingFace models [1] [2]. Its README shows a simple Python example of defining a function schema and generating a compliant function call. Good for learning how to constrain open-source models to output JSON.
- **AutoGen by Microsoft (Tools)** – Demonstrates a code execution tool running in a Docker sandbox [43] and how `FunctionTool` automatically generates JSON schemas [3]. Great for seeing an advanced agent toolkit that emphasizes security (Docker) and ease of use (auto schema).
- **Hugging Face *smolagents* (Secure Code Execution)** – Tutorial on a custom safe Python interpreter [22] [23]. Provides code snippets showing how unauthorized imports or infinite loops are blocked. This resource is useful for implementing fine-grained code safety within Python itself (particularly cross-platform, since it doesn't rely on OS-specific sandboxing).
- **Auto-GPT Documentation** – Sections on the workspace and command restrictions [27] [33]. Illustrates how a popular agent limits file access to a sandbox folder and allows disabling dangerous commands via config. It also has a configurable `shell_command_control` with allow/deny lists [20]. This is a practical reference for safe file operations and user-facing config toggles.
- **OpenAI Cookbook – Streaming Completions** – Shows how to use `stream=True` in the OpenAI API and iterate over response chunks [44] [9]. Although geared to OpenAI's service, the same pattern applies to local endpoints supporting SSE streaming. It's a quick way to learn streaming if you haven't used it before.
- **LangChain Discussions (Concurrency)** – Community example of using `asyncio.gather` to run multiple tools in parallel within an agent [13]. This is a short snippet that succinctly demonstrates the core idea of parallel tool execution.
- **Bret Cameron's Blog on Token Limits** – Explains strategies to break the context limit barrier, like rolling windows and summarization [36]. It's not code-heavy, but it gives insight into long-term memory approaches for chatbots. If your agent will engage in long dialogues or analyze long documents, this is a must-read for design ideas.

By combining the techniques and patterns above, you can build a capable local AI agent that calls tools safely and efficiently. Always test your loop thoroughly with different scenarios (normal queries, edge cases, malicious inputs) to ensure the safeguards hold up. With a high-performing model like Qwen-3 and the right tooling around it, you'll have an AI agent that can not only chat, but act – all under your own roof and rules. Good luck, and build responsibly!

[1] [2] GitHub - rizerphe/local-llm-function-calling: A tool for generating function arguments and choosing what function to call with local LLMs
https://github.com/rizerphe/local-llm-function-calling

[3] [17] [18] [19] [43] Tools — AutoGen
https://microsoft.github.io/autogen/stable//user-guide/core-user-guide/components/tools.html

[4] [5] [6] [7] [8] Tool Use | LM Studio Docs
https://lmstudio.ai/docs/app/api/tools

[9] GitHub - BerriAI/litellm: Python SDK, Proxy Server (LLM Gateway) to call 100+ LLM APIs in OpenAI format - [Bedrock, Azure, OpenAI, VertexAI, Cohere, Anthropic, Sagemaker, HuggingFace, Replicate, Groq]
https://github.com/BerriAI/litellm

[10] [11] [12] Tool Use _ LM Studio Docs | PDF
https://www.scribd.com/document/871946525/Tool-Use-LM-Studio-Docs

[13] [14] Can we use multiple tools at once in AgentExecutor? · langchain-ai langchain · Discussion #21869 · GitHub
https://github.com/langchain-ai/langchain/discussions/21869

[15] [16] [26] [29] [39] GitHub - realugbun/AgentExecMCP: Secure Shell and Code Execution Sandbox for AI Agents
https://github.com/realugbun/AgentExecMCP

[20] [21] [30] [32] Built in Components - AutoGPT Documentation
https://docs.agpt.co/forge/components/built-in-components/

[22] [23] [24] [25] Secure code execution
https://huggingface.co/docs/smolagents/main/en/tutorials/secure_code_execution

[27] [28] [33] [35] Usage - AutoGPT Documentation
https://docs.agpt.co/classic/usage/

[31] Settings Management - Pydantic
https://docs.pydantic.dev/latest/concepts/pydantic_settings/

[34] OpenAI Compatibility API | LM Studio Docs
https://lmstudio.ai/docs/app/api/endpoints/openai

[36] 3 Strategies to Overcome OpenAI Token Limits | Bret Cameron
https://www.bretcameron.com/blog/three-strategies-to-overcome-open-ai-token-limits

[37] [38] Running agents
https://langchain-ai.github.io/langgraph/agents/run_agents/

[40] Best practices for prompt engineering with the OpenAI API
https://help.openai.com/en/articles/6654000-best-practices-for-prompt-engineering-with-the-openai-api

[41] [42] Function Calling - Qwen
https://qwen.readthedocs.io/en/latest/framework/function_call.html

[44] How to stream completions
https://cookbook.openai.com/examples/how_to_stream_completions