ChatGPT

Building a Local Qwen3 Codex-Style Agent with MCP Tool-Calling

Setting Up a Local Qwen3 Model Service

To integrate Qwen3 into a local tool-using agent, first deploy the Qwen3 model with an OpenAI-compatible API endpoint. Qwen3 supports multiple local serving backends, including **vLLM**, **Ollama**, and **LM Studio** [1] [2]. For example, you can use **Ollama** to quickly run a quantized Qwen3 model on your machine:

- **Install Ollama:** Download and install Ollama (supports GGUF quantized Qwen3 models) [3] [4].
- **Launch Qwen3 in Ollama:** Run `ollama run qwen3:8b` (for the 8B model) or another variant. Ollama will download the model and host it at a local endpoint (default `http://localhost:11434`) [5]. This endpoint speaks the OpenAI Chat API protocol.
- **Alternatively, use vLLM:** Install `vllm>=0.8.5` and start a server. For example, `vllm serve Qwen/Qwen3-8B --host 0.0.0.0 --port 8000` will serve an OpenAI-compatible API on `localhost:8000` [6] [7]. (For Qwen3 models, enable reasoning mode but *disable* auto tool execution flags if using Qwen-Agent to handle tools [1].)

LM Studio can also load Qwen3 quantized models (e.g. GGUF format) for local use [8] [9], though it primarily provides a UI. To use it as a backend service, you would integrate via its API support (for example by enabling its **OpenAI-compatible server** mode if available, or by running Qwen3 in a library like llama.cpp which LM Studio supports). In practice, vLLM or Ollama are recommended for exposing a stable OpenAI-style API for Qwen3 [6] [10].

Once running, verify the API is accessible. For instance, with vLLM at port 8000, a test completion can be made via curl or the OpenAI Python SDK [11] [12]. With Ollama, the API base is `http://localhost:11434/v1`. Keep the model server running for the next steps.

Initializing Qwen-Agent with a Local Model

Alibaba's **Qwen-Agent** framework provides the logic for function calling, tool use, planning, and memory on top of Qwen models [13]. We'll use its high-level `Assistant` agent to handle our Codex-style loop. First, install Qwen-Agent with all extras:

```
pip install -U "qwen-agent[gui,rag,code_interpreter,mcp]"
```

This includes support for the web GUI, Retrieval-Augmented Generation, the Code Interpreter tool, and MCP integration [14].

Now configure the agent to use your local Qwen3 service. In Qwen-Agent, the model is specified via an `llm_cfg` dict. For a self-hosted model, set the `model_server` to your API endpoint and provide a dummy API key (if none is required). For example, to use an Ollama-served Qwen3 8B model on port 11434:

```
llm_cfg = {
    'model': 'qwen3:8b',
    'model_server': 'http://localhost:11434/v1',  # OpenAI-compatible base URL
(Ollama)
    'api_key': 'EMPTY',  # not needed for local but Qwen-Agent expects a key
    'generate_cfg': { 'top_p': 0.8 }  # optional generation params
}
```

[10] If using vLLM on port 8000, use `'model_server': 'http://localhost:8000/v1'` and the model name from your deployment (e.g. `"Qwen/Qwen3-8B"` for the HF model ID) [15] [16]. Qwen-Agent also provides a helper `get_chat_model()` that can wrap a raw OpenAI API endpoint with Qwen3's function-calling capabilities [17] [18].

Finally, instantiate the agent. Qwen-Agent's `Assistant` class takes the LLM config, and optional system message, tools, or files. For example:

```
from qwen_agent.agents import Assistant

agent = Assistant(llm=llm_cfg, system_message="You are a coding assistant...",
function_list=tools)
```

At this point, the agent will connect to your local model and be ready to manage function calls and tools as described below.

Defining Custom JSON-Schema Tools in Qwen-Agent

Qwen-Agent allows you to register custom tools (functions) with JSON Schema definitions for their parameters, similar to OpenAI function calling. You can create new tools by subclassing `BaseTool` and decorating with `@register_tool("tool_name")`. For example, to add a tool that executes git status:

```
import json, subprocess
from qwen_agent.tools.base import BaseTool, register_tool

@register_tool('git_status')
class GitStatusTool(BaseTool):
    """Report the current Git repository status."""
    description = "Get the current Git repository status in the working
directory."
    parameters = []  # no parameters for this tool
```

```
    def call(self, params: str, **kwargs) -> str:
        # Run `git status` and return the output as JSON
        result = subprocess.run(["git", "status"], capture_output=True,
text=True)
        return json.dumps({"status": result.stdout}, ensure_ascii=False)
```

This declares a tool named `"git_status"` with no input fields and returns a JSON string result. The `description` tells the model what the tool does, and `parameters` defines a JSON Schema for inputs (here empty) [19] [20]. When the agent is initialized, it will include this tool in its available function list. The model can then decide to call `git_status` if a user asks something requiring it (e.g. "show my git status").

For a more complex example, Qwen-Agent's documentation provides an **image generation** tool. It defines an input parameter (`prompt: string`) and returns an image URL:

```
@register_tool('my_image_gen')
class MyImageGen(BaseTool):
    description = 'AI image generation service: input a text prompt, get an
image URL.'
    parameters = [{
        'name': 'prompt', 'type': 'string', 'description': 'Description of the
desired image', 'required': True
    }]
    def call(self, params: str, **kwargs) -> str:
        prompt = json5.loads(params)['prompt']
        # ... call an image API or service ...
        return json5.dumps({'image_url': f'https://image.pollinations.ai/prompt/
{prompt}'}, ensure_ascii=False)
```

[21] [22]. This tool would let the model request an image by providing a text prompt.

Built-in Tools: Qwen-Agent comes with some built-ins like `"code_interpreter"` (a Python execution sandbox) and web browsing tools [23]. You can include these by name in the agent's `function_list`. For instance, `tools = ['my_image_gen', 'code_interpreter']` would enable the custom image tool and the code interpreter [24]. *Note:* The code interpreter executes Python code in your environment without sandboxing (intended for local testing only) [25], so use caution if exposing it.

Each tool ultimately gets presented to the model as a function with a JSON schema. Under the hood, Qwen3 expects the tool list in the format:

```
[
  {
    "type": "function",
    "function": {
      "name": "<tool_name>",
```

```
      "description": "<tool_description>",
      "parameters": { ... JSON Schema ... }
    }
  },
  ...
]
```

[26] [27]. Qwen-Agent handles this formatting for you when you supply `function_list` to the Assistant.

Integrating MCP Servers (External Tools via JSON-RPC)

One powerful feature is using **Model-Context-Protocol (MCP)** servers as tools. MCP servers are external processes (Node.js or Python) that implement tools (functions or data sources) following a standard protocol (think of it like JSON-RPC for LLM tools). Qwen3 is designed to leverage MCP for things like database queries, web browsing, memory storage, etc. [28].

Example – SQLite Database Query: Qwen-Agent provides an example `assistant_mcp_sqlite_bot.py` using an MCP server for SQLite. The idea is to spin up an MCP tool server that can execute SQL queries on a local database, and have the agent route user requests to it. Here's how you can declare an MCP-based tool in Qwen-Agent:

```
tools = [{
    "mcpServers": {
        "sqlite": {
            "command": "uvx",
            "args": ["mcp-server-sqlite", "--db-path", "test.db"]
        }
    }
}]
agent = Assistant(llm=llm_cfg, function_list=tools, name="MCP-SQLite-Bot",
                  description="This bot can answer questions by querying a
SQLite DB")
```

[29] [30]. In this configuration: - We use the special key `"mcpServers"` in the tools list. We define a server named `"sqlite"` to handle our database tool. - The `command` `"uvx"` is an **executor** (provided by the `uv` tool) that runs the MCP server. Here `uvx` will fetch and run the `mcp-server-sqlite` tool in an isolated environment if needed [31] [32]. - `args` specify launching the SQLite MCP server with the path to the database (`--db-path test.db`).

When the agent starts, it will spawn this subprocess. The MCP server registers a *tool* (or a set of tools) that it can handle. Qwen-Agent's MCP manager will call the server's `tools/list` endpoint to discover its functions [33] [34]. In this case, the SQLite server typically provides a function (e.g. `query`) to execute SQL on the DB. Qwen-Agent will map that to the tool name "sqlite" for the model to use.

Running MCP Servers: To use MCP tools, ensure you have the prerequisites installed: **Node.js**, **Git**, and the Python `uv` package (version ≥0.4.18) [35] [36]. The `uv` tool is used to seamlessly run MCP servers; for example `uvx mcp-server-sqlite` will automatically fetch the MCP SQLite server package if not present and launch it [37] [32]. (You can think of `uvx` as analogous to `pipx run` for MCP servers). The SQLite server in this example is likely implemented in Node.js, hence the need for Node. Once launched, Qwen-Agent communicates with the MCP server via stdio or HTTP (as configured) to send tool requests and receive results.

You can add **multiple MCP servers** in the config. For example, you might have another entry for `"git"` or `"filesystem"`:

```
tools = [{
    "mcpServers": {
        "sqlite": { ... },
        "git":    { "command": "uvx", "args": ["mcp-server-git"] }
    }
}]
```

Each server can expose one or more tools. Qwen-Agent will register all tools from all MCP servers so the model can choose among them. This is how Qwen3 can interface with a variety of tools and contexts dynamically [28].

Safety and Permissions: MCP tools are intended to run with a human in the loop for approval [38]. In practice, you might design your MCP servers and agent UI such that certain actions (like file writes or external calls) ask for user confirmation. The MCP specification provides *annotations* for tools (like `readOnlyHint` or `destructiveHint`) to signal their safety characteristics [39]. For instance, a file-editing tool would be marked as destructive (since it modifies data) and an agent UI could require the user to approve its use. When building custom MCP servers or tools, consider these best practices to maintain a safe execution environment.

Multi-Turn Function Calling and Tool Use Loop

With the model, agent, and tools in place, the core loop resembles the classic OpenAI function-calling workflow: the model may output a function call, the agent executes it, feeds the result back, and the model continues. Qwen3 is capable of **multi-step reasoning**, meaning it can plan and invoke multiple tools in sequence (even in one user query) before producing a final answer [40] [41].

Qwen-Agent provides a structured interface for this. When you call `agent.run(messages)`, it will stream the model's responses. Under the hood, Qwen-Agent parses the model output and intercepts function calls. In **"no thinking" mode**, the model directly emits one or more `function_call` entries; in **"thinking" mode**, it emits a `reasoning_content` (chain-of-thought) first, then the function calls [42] [43]. For example, Qwen3 might output two function calls in a row if the task requires two tools (as seen in a weather query example, where it called `get_current_temperature` then `get_temperature_date`) [44] [45].

The agent (or your integration code) should loop through these calls in order, execute each, and append the results to the conversation. A simplified code pattern for the loop is given in the Qwen docs:

```
for message in responses:
    if fn_call := message.get("function_call"):
        fn_name = fn_call['name']
        fn_args = json.loads(fn_call["arguments"])
        result = get_function_by_name(fn_name)(**fn_args)  # execute the tool
        fn_res = json.dumps(result)
        messages.append({
            "role": "function", "name": fn_name, "content": fn_res
        })
```

[46] [47]

After inserting the `{"role": "function", ...}` message with the tool's output, you then call the model again (continuing the `agent.run` loop) to let it consume the result and either call another tool or produce an answer [48] [49]. Qwen-Agent automates much of this logic when you use `Assistant.run()`. It will yield the assistant's messages, including final answers or any number of tool invocation steps, updating the message history as needed.

Notably, Qwen3 (especially larger variants like **QwQ-32B**) can handle parallel and multi-turn tool usage plans [40]. In practice this means the model might decide to call several functions in one thought, or iterate tool calls over multiple turns to refine an answer. Qwen-Agent's function call template and parsing are designed to support these advanced patterns. The **"reasoning parser"** (often configured as `deepseek_r1`) helps extract structured tool calls even when the model intermixes them with reasoning text [50].

Best Practices for MCP Tool Integration

When building a Codex-style agent with Qwen3 and MCP, keep in mind these tips drawn from the official SDK and examples:

- **Design Tools with Clear Schemas:** Ensure each tool's JSON schema precisely defines inputs and that the description is clear. Qwen3 will use this to decide when and how to call the function [51]. If your tool has complex inputs, provide enums, required fields, and examples if possible. This improves the model's tool-calling accuracy.
- **Use Descriptive Names:** Name your tools for their function (e.g. `"python_exec"` for a code runner, `"file_search"` for a file-finder). Avoid names that overlap or confuse the model. The MCP tool definition includes a human-readable `title` and `description` — the `title` is optional, and Qwen-Agent will strip out any extraneous `title` fields from schemas to avoid OpenAI API compliance issues [52] [53].
- **Leverage MCP Annotations:** The Model Context Protocol allows tagging tools with metadata about side effects. For instance, set `readOnlyHint: true` for a tool that only reads data, or `destructiveHint: true` for one that alters external state [39]. This can guide the agent (and

user) in understanding the risk. In a safe file-editing scenario, you might expose both a read-only file viewer tool and a separate write tool marked destructive, so the model uses them appropriately.

- **Human Approval for Critical Actions:** Always keep a human in the loop for potentially dangerous operations. MCP is built around the idea that the AI suggests tool use, but a person grants final execution [38]. If you build a UI (or even a CLI prompt loop), consider confirming with the user before executing `python_exec` code or modifying files. This is especially important since Qwen-Agent's code interpreter is not sandboxed [25].
- **Persistent Memory via MCP:** Qwen3 can use tools for long-term memory. For example, an MCP "memory" server could store and retrieve conversation context on disk or a database. This would be configured similarly (perhaps a `"memory": {"command": "uvx", "args": ["mcp-server-memory"]}` tool). When implementing such memory, ensure the content is indexed or scoped so the model doesn't fetch irrelevant data. Qwen3's prompt or system message can instruct it when to use the memory tool (e.g. on user requests like "remember where we left off").
- **Client-Server Integration:** If writing your own MCP server (using the TypeScript SDK or others), follow the MCP spec for **tool list** and **tool call** handlers. Define your tools on the server with unique `name` and proper input schema, and implement the `ListTools` and `CallTool` request handlers as shown in the SDK examples [54] [55]. This ensures Qwen-Agent (as the MCP client) can discover and invoke them seamlessly. The SDK takes care of the transport (stdio, HTTP, etc.), so you just register functions.
- **Dynamic Routing:** The term "dynamic tool routing" means the agent dynamically decides which tool server to call based on the function name. In our setup, Qwen-Agent's `MCPManager` will route calls to the correct MCP server automatically once tools are registered. You just need to ensure each tool name is unique across your servers. For instance, if two servers both provide a tool called `"search"`, differentiate them (perhaps one as `"web_search"` and another as `"local_search"`). This way, the model can specify the correct one and Qwen-Agent will route to the appropriate server process.
- **Offline Operation:** All these components can run fully offline. Qwen3 models are available in local formats (HF Transformers, GGUF, etc.), and the MCP servers like `mcp-server-sqlite` or others run locally (they may pull necessary packages the first time via `uv`, so do that in an internet-enabled environment, or pre-download the packages). Once set up, the model's reasoning and tool use loop requires no cloud services – as demonstrated by running Qwen3 on an **AI PC** with Qwen-Agent and Ollama in a recent guide [56] [57].

By following these patterns and examples, you can adapt your `codex_local_shell_bridge.py` to create a powerful local AI agent. The agent will use Qwen3's advanced function-calling to execute code, check git status, edit files safely, and otherwise act like an offline version of Codex with tools – all while citing the structured JSON schemas and protocols that make this integration robust.

Sources: The code and configurations above were drawn from Qwen's official examples and documentation, including the Qwen-Agent repository [29] [10], Qwen3 function-calling docs [46], and the Model Context Protocol SDK guidelines [39] [54]. These resources provide further detail for customizing and extending your setup.

---

[1] [6] [13] [40] [50] GitHub - QwenLM/Qwen-Agent: Agent framework and applications built upon Qwen>=3.0, featuring Function Calling, MCP, Code Interpreter, RAG, Chrome extension, etc.
https://github.com/QwenLM/Qwen-Agent

2 3 5 10 21 22 56 57 Deploying AI Agents Locally with Qwen3, Qwen-Agent, and Ollama | by Benjamin Consolvo | Intel Tech | May, 2025 | Medium
https://medium.com/intel-tech/deploying-ai-agents-locally-with-qwen3-qwen-agent-and-ollama-cad452f20be5

4 How to Use Qwen3 Quantized Models Locally: A Step-by-Step Guide
https://apidog.com/blog/qwen3-quantized-models-locally/

7 11 12 vLLM - Qwen
https://qwen.readthedocs.io/en/latest/deployment/vllm.html

8 Qwen on X: "We're officially releasing the quantized models of ...
https://x.com/Alibaba_Qwen/status/1921907010855125019

9 Qwen releases official quantized models of Qwen3 - Threads
https://www.threads.com/@harsha_gadekar/post/DJjuvp0p_gd/qwen-releases-official-quantized-models-of-qwen3now-users-can-deploy-qwen3-via-o

14 28 29 30 31 32 35 36 37 Qwen 3 Has MCP Server Support and Here's How to Use It
https://apidog.com/blog/qwen-3-mcp-server/

15 16 17 18 26 27 41 42 43 44 45 46 47 48 49 51 Function Calling - Qwen
https://qwen.readthedocs.io/en/latest/framework/function_call.html

19 20 23 24 25 raw.githubusercontent.com
https://raw.githubusercontent.com/QwenLM/Qwen-Agent/main/README_CN.md

33 34 52 53 mcp tool bug · Issue #475 · QwenLM/Qwen-Agent · GitHub
https://github.com/QwenLM/Qwen-Agent/issues/475

38 39 54 55 Tools - Model Context Protocol
https://modelcontextprotocol.io/docs/concepts/tools