Qwen Function Calling: Official Guidance and Your 0.6B Challenge

Your problem is a documented limitation. Qwen's 0.6B model lacks the reasoning capacity for reliable multi-step tool calling—it's not a prompt structure issue you can easily fix. The official documentation, research papers, and community testing all confirm that models this small struggle with the cognitive demands of multi-step agentic workflows.

The core issue: Small models can't think ahead

Qwen3-0.6B was trained using "strong-to-weak distillation" from larger models, focusing primarily on supervised fine-tuning rather than the full reinforcement learning pipeline that teaches larger models complex reasoning. The official technical report acknowledges this creates models "very strong on benchmarks, but potentially less robust on a wider domain of tasks." Multi-step tool calling—where the model must plan, execute, then recognize incompleteness—falls squarely into this limitation.

Research confirms this isn't unique to Qwen. The EMNLP 2024 paper "Small LLMs Are Weak Tool Learners" demonstrates that smaller models fundamentally struggle with planning and multi-agent workflows. The Qwen documentation explicitly warns: "It is not guaranteed that the model generation will always follow the protocol even with proper prompting or templates, especially for templates that are more complex and rely more on the model itself to think and stay on track."

1. Qwen's recommended format: Hermes-style with XML tags

Qwen3 officially recommends **Hermes-style tool calling** using XML tag delimiters. This format is built directly into the model's `tokenizer_config.json` and provides the best performance.

System message structure:

```
<|im_start|>system
You are Qwen, created by Alibaba Cloud. You are a helpful assistant.

# Tools

You may call one or more functions to assist with the user query.

You are provided with function signatures within <tools></tools> XML tags:
<tools>
{"type": "function", "function": {"name": "get_current_weather", "description":
"Get the current weather in a given location.", "parameters": {...}}}
</tools>

For each function call, return a json object with function name and arguments
within <tool_call></tool_call> XML tags:
<tool_call>
{"name": <function-name>, "arguments": <args-json-object>}
</tool_call>
<|im_end|>
```

Function definition format follows OpenAI's schema:

python

```python
tools = [
    {
        "type": "function",
        "function": {
            "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
                    },
                    "unit": {
                        "type": "string",
                        "enum": ["celsius", "fahrenheit"]
                    }
                },
                "required": ["location"]
            }
        }
    }
]
```

The key XML tags are `<tools></tools>` for function definitions and `<tool_call></tool_call>` for the model's invocations. For Qwen-Agent framework users, set `fncall_prompt_type: 'nous'` (the default for Qwen3).

2. Templates for smaller models: No special accommodation exists

Qwen provides no specific templates or simplified formats for smaller models like 0.6B. All models use the same Hermes-style template, which assumes the model has sufficient reasoning capacity to understand and follow the protocol.

This is problematic because the template's complexity requires the model to "think and stay on track"—precisely what smaller models struggle with. Community evidence supports this: a developer created "Pelmeshek/qwen3-0.6B-function-calling-lora," a fine-tuned variant specifically for function calling, suggesting the base 0.6B model's capabilities were insufficient.

Docker's practical evaluation of local models for tool calling found that smaller models (under 10B parameters) exhibit recurring failures: **eager invocation** (calling tools for simple greetings), **wrong tool selection**, and **invalid arguments**. One tester noted: "Model this small needs both a short prompt and some very specific wording."

3. Multi-step recommendations: The while loop pattern

Qwen's official guidance for multi-step scenarios uses an **iterative message-based loop** that continues until the model returns a final answer without tool calls.

The canonical pattern from Alibaba Cloud documentation:

```python
def multi_step_tool_calling(messages):
    # Step 1: Initial request
    response = client.chat.completions.create(
        model="qwen-plus",
        messages=messages,
        tools=tools
    )

    assistant_output = response.choices[0].message
    messages.append(assistant_output)

    # Step 2: While loop - continue until no tool_calls
    while assistant_output.tool_calls != None:
        # Execute all tool calls
        for tool_call in assistant_output.tool_calls:
            function_result = execute_function(
                tool_call.function.name,
                json.loads(tool_call.function.arguments)
            )

            # Step 3: Add tool result with matching ID
            messages.append({
                "role": "tool",
                "content": function_result,
                "tool_call_id": tool_call.id
            })

        # Step 4: Get next response
        assistant_output = client.chat.completions.create(
            model="qwen-plus",
            messages=messages,
            tools=tools
        ).choices[0].message

        messages.append(assistant_output)

    return assistant_output.content
```

Critical details for tool result handling:

1. **Tool call ID matching**: Each tool result MUST include the `tool_call_id` that exactly matches the original call's ID
2. **Role specification**: Tool results use `role: "tool"` (not "function")
3. **Content serialization**: Return results as strings, JSON-serialized if complex
4. **Remove tool_choice on final turn**: Don't set `tool_choice` parameter when expecting the final answer, or the model will return more tool calls instead of summarizing

System prompt guidance improves reliability: "Emphasizing when to call the tools in the System Message can improve the accuracy of function calling." Example:

```python
{
    "role": "system",
    "content": """You are a helpful assistant that can access external functions.
    The responses from these function calls will be appended to this dialogue.
    Please provide responses based on the information from these function calls.

    If you need weather information, call get_current_weather.
    If you need forecast information, call get_weather_forecast.
    Continue using tools until you have all information needed to answer completel
}
```

4. Specific quirks and limitations for 0.6B models

The documentation doesn't sugarcoat this: **0.6B models are inadequate for multi-step tool calling.** Here's what makes them problematic:

Training methodology differences: Smaller Qwen3 models (0.6B, 1.7B, 4B) received "strong-to-weak distillation" training from larger models—primarily supervised fine-tuning on synthetic data. They did NOT receive the full 4-stage training pipeline that larger models got, which includes Long CoT Cold Start, Reasoning RL, Thinking Mode Fusion, and General RL. This distillation approach creates capability gaps.

Documented performance issues:

- "Subpar performance in coding-related tests" (official Qwen3 analysis)
- Context limited to 32K tokens (versus 128K for 8B+)
- Function calling implemented via prompt engineering, not specialized training—meaning reliability depends entirely on the model's ability to follow complex instructions
- Official warning: "It is not guaranteed that the model generation will always follow the protocol"

Multi-step planning deficits: The vLLM documentation notes that "the quality of multi-step tool use with Qwen3 thinking models may be suboptimal" due to preprocessing that drops reasoning content fields. For a 0.6B model without sophisticated reasoning capabilities to begin with, this compounds the problem.

Research evidence: The academic paper "Improving Small-Scale Large Language Models Function Calling for Reasoning Tasks" (arXiv:2410.18890) exists precisely because small models struggle with function calling. It introduces specialized training frameworks to address these limitations.

Minimum recommended size: For reliable multi-step tool calling, Qwen recommends **7B+ parameters**. Specifically:

- Simple, single-step calls: 1.5B-3B may work with careful prompting
- Multi-step scenarios: Qwen2.5-7B or Qwen3-8B minimum
- Production-grade agentic workflows: 14B+ or MoE models (Qwen3-30B-A3B)

5. Handling tool results and chaining: Message structure is everything

Tool result format must be exact:

```python
messages.append({
    "role": "tool",
    "content": json.dumps({"temperature": "72F", "condition": "sunny"}),
    "tool_call_id": "call_abc123"  # Must match original call ID
})
```

Parallel versus sequential calling: Qwen supports both, with different use cases:

- **Parallel** (default, enabled with `parallel_tool_calls=True`): Get all needed tool calls at once when tasks have no dependencies. Example: checking weather in multiple cities simultaneously.
- **Sequential** (while loop pattern): Required when "dependencies exist between tasks—for example, the input for Tool A depends on the output of Tool B." This is your weather→forecast scenario.

The complete message flow for your specific use case should look like:

```python
messages = [
    {"role": "system", "content": "You are a helpful assistant. Use get_weather fo
    {"role": "user", "content": "What's the weather now and forecast for tomorrow
]

# Round 1: Model calls get_weather
messages.append({
    "role": "assistant",
    "content": "",
    "tool_calls": [{"id": "call_1", "function": {"name": "get_weather", "arguments
})

# Round 1 result
messages.append({
    "role": "tool",
    "content": '{"temp": "65F", "condition": "cloudy"}',
    "tool_call_id": "call_1"
})

# Round 2: Model should call get_forecast
# (But 0.6B models often stop here instead)
```

The problem your 0.6B model faces: After receiving the first tool result, it lacks the reasoning capacity to:

1. Recognize the original query asked for TWO pieces of information
2. Understand it has only retrieved ONE
3. Determine it needs to make a SECOND tool call
4. Generate the appropriate next tool invocation

This isn't a prompt structure issue—it's a **cognitive limitation** of the model size.

Practical solutions for your situation

Option 1: Upgrade to Qwen3-8B or larger (recommended) The performance gap is substantial. Qwen3-8B has 128K context, full reasoning training, and demonstrated multi-step capabilities. The reliability improvement will save you debugging time that exceeds the computational cost difference.

Option 2: External orchestration logic Don't rely on the 0.6B model for multi-step planning. Implement application-level logic:

```python
def handle_weather_and_forecast(location):
    # Explicit orchestration - you control the steps

    # Step 1: Always get current weather
    weather_result = get_weather(location)

    # Step 2: Always get forecast
    forecast_result = get_forecast(location)

    # Step 3: Give both to model for summarization only
    messages = [
        {"role": "user", "content": f"Summarize this weather data: Current: {weath
    ]

    response = client.chat.completions.create(
        model="qwen3:0.6b",
        messages=messages
        # No tools parameter - just summarization
    )

    return response.choices[0].message.content
```

Option 3: Fine-tune for specific workflow Following the community example of function calling LoRAs, create a fine-tuned adapter specifically for your weather→forecast pattern. This teaches the model the exact sequence you need.

Option 4: Hybrid approach Use 0.6B for initial tool selection, but a larger model (or deterministic logic) for multi-step planning:

python

```python
# 0.6B: "Do I need weather tools for this query?"
if needs_weather_tools(user_query, small_model):
    # Application logic: "If weather, always get both"
    results = [get_weather(loc), get_forecast(loc)]
    # 0.6B: "Summarize these results"
    return summarize(results, small_model)
```

The bottom line: Your problem is architectural, not configurational

Qwen's official documentation provides excellent guidance for function calling, but it assumes model capabilities that 0.6B simply doesn't have. The Hermes-style templates, while optimal, are still complex protocols that require reasoning capacity. Your specific issue—the model thinking the task is complete after one tool call—is a **documented limitation of small models' planning abilities**, not a problem you can solve by adjusting prompt structure.

The documentation's warning applies directly to your case: "For production code, be prepared that if it breaks, countermeasures or rectifications are in place." For 0.6B models doing multi-step tool calling, "if" is really "when."

If you must continue with 0.6B, implement external orchestration logic that doesn't depend on the model's multi-step reasoning. Otherwise, upgrade to Qwen3-8B minimum—the first model in the Qwen3 family with 128K context and the full training pipeline that enables reliable multi-step agentic workflows.