GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN IN PUBLICA COMMODA SEIT 1737

HPS

Seminar Report

Open source Agents

Aayush Gupta

MatrNr: 20076685

Supervisor: Michael Bidollahkhani

Georg-August-Universität Göttingen
Institute of Computer Science

March 21, 2026

Abstract

This report investigates reliable tool calling in open-source LLM agents for a weather-assistant use case. The main challenge is robust multi-step execution when the model must call external APIs in the correct order and with valid arguments. Two implementations are compared under the same API interface: a vanilla orchestration tool calling pipeline implementation without the use of agentic frameworks and a LangChain-based structured tool-calling pipeline. Both variants integrate geocoding, weather retrieval, and optional SMS dispatch. The vanilla variant is also deployed as a public web service on a DigitalOcean droplet to evaluate practical deployment constraints beyond local experiments. Evaluation is performed through repeated HTTP benchmarks across four open-source models using completion rate, tool accuracy, argument accuracy, step validity, and latency. The results indicate that structured tool calling improves end-to-end reliability and sequence correctness compared to vanilla orchestration implementation for most tested models.

Keywords: open-source LLM agents, tool calling, LangChain, FastAPI, weather assistant, benchmarking

Declaration on the use of ChatGPT and comparable tools in the context of examinations

In this work I have used ChatGPT or another AI as follows:

☐ Not at all

☐ During brainstorming

☑ When creating the outline

☐ To write individual passages, altogether to the extent of 0% of the entire text

☐ For the development of software source texts

☐ For optimizing or restructuring software source texts

☑ For proofreading or optimizing

☐ Further, namely:

I hereby declare that I have stated all uses completely.
Missing or incorrect information will be considered as an attempt to cheat.

Contents

List of Tables iv

List of Figures iv

List of Abbreviations v

1 Introduction 1

1.1 Problem Statement . . . 1

1.2 Objective of the Project . . . 2

1.3 Contributions . . . 2

2 Background and Related Work 2

2.1 Related work . . . 3

3 Use Case and framework overview 3

3.1 Why agent is needed . . . 3

3.2 Framework overview . . . 4

4 Implementation 4

4.1 Vanilla Architecture . . . 5

4.2 LangChain Architecture . . . 6

4.3 Vanilla Hosting Architecture on DigitalOcean Droplet . . . 8

5 Evaluation 10

5.1 Models . . . 10

5.2 Parameter used for Evaluation . . . 11

5.3 Prompts used for Evaluation . . . 12

5.4 Results for Vanilla implementation . . . 13

5.5 Results for LangChain implementation . . . 14

5.6 Evaluation Challenges (reproducibility, correctness verification) . . . 14

6 Discussion 15

7 Conclusion 16

References 17

A Work Sharing A1

B Additional Notes A1

List of Tables

1 Vanilla implementation results (comparable models) 13
2 LangChain implementation results (comparable models) 14

List of Figures

1 Vanilla orchestration flow with parser-mediated tool chaining. 6
2 LangChain orchestration flow with structured tool-calling loop. 7
3 Deployed website interface. 8
4 Sample SMS generated by the deployed system. 9
5 Vanilla deployment architecture on DigitalOcean droplet. 10

List of Abbreviations

API Application Programming Interface

LLM Large Language Model

SMS Short Message Service

1 Introduction

Large Language Models (LLMs) have brought a new revolution to computer science. However, traditional LLMs are inherently limited by their training cutoff and by their inability to interact with the outside world. As a result, LLMs operate as isolated and stateless systems. AI agents are the next step in this process. The core idea is to give LLMs access to external tools such as APIs, databases, memory mechanisms, and iterative reasoning loops. In this way, LLMs can evolve from conversational bots into autonomous problem solvers capable of executing multi-step tasks [1].

The incredible progress in the open-source ecosystem, especially in the area of LLMs, has created an opportunity for broader public adoption of agentic systems rather than limiting them to large companies. Open-weight models such as LLaMA [2], Mistral [3], and Qwen [4] have made it easier than ever to integrate LLMs with external tools using frameworks such as LangChain [5] and AutoGen [6].

It serves two purposes. First, it builds an agentic system without using frameworks such as LangChain [5], in order to understand the fundamental mechanisms of tool calling, output parsing, and orchestration that frameworks usually abstract away. Second, it implements the same project with LangChain [5] and compares both versions using five evaluation metrics and four open-source LLMs of different sizes.

The goal is not to compare all available frameworks, but to show how much a framework helps and to compare open-source models of varying sizes. The models used here are not necessarily the strongest models available in the market; they were selected because they are easier and cheaper to host. These models can be hosted locally on currently available consumer hardware.

An additional part of this project involves deploying the agent as a publicly accessible web service on a DigitalOcean virtual machine. This adds a real-world systems dimension to the work: the agent is not just a notebook experiment, but a deployed service with reverse proxy, process management, firewall configuration, and a browser-based chat interface. This was done to learn and understand the end-to-end process of hosting agents.

1.1 Problem Statement

Building an LLM-based agent that reliably calls external tools in the correct order, with correct arguments, across multiple conversation turns is a difficult engineering challenge. This becomes especially hard when no framework is used, because the LLM output must be manually parsed to match tool-execution requirements. This manual parsing is fragile, since LLMs often produce varying formats and their output is not always consistent or predictable.

This challenge becomes even more complex when transitioning from a local environment to a real production environment, where system complexity rises and there is little

room for unpredictability. In such situations, unpredictable agent behavior can lead to operational failures.

Agentic frameworks such as LangChain [5] attempt to solve this problem by providing structured tool-calling interfaces, but they also introduce their own complexity and dependencies.

1.2 Objective of the Project

1. Build an agent that can call multiple tools based on a user request, without using frameworks. The agent should handle weather queries through a three-tool chain: obtain geographic coordinates for a city, retrieve weather information using those coordinates, and send a text message to the user with the weather information and one practical tip.

2. Deploy the vanilla agent as a publicly accessible web service on a DigitalOcean Ubuntu droplet, moving it from a local PC to a globally reachable deployment.

3. Rebuild the same agent using LangChain, while keeping the API contract and evaluation interface identical so both implementations can be benchmarked using the same scripts. 4. Evaluate both implementations across open-source LLMs using a structured prompt suite and quantitative metrics covering completion rate, tool accuracy, argument accuracy, step validity, and latency.

1.3 Contributions

The contributions of this work are as follows:

1. A fully documented vanilla agent implementation that makes tool-calling orchestration explicit. It serves as a baseline for understanding tool calling in agents.

2. A production deployment architecture on DigitalOcean that demonstrates how to take an LLM agent from a local script to a publicly accessible service.

3. A quantitative evaluation comparing four open-source LLMs across two approaches, with defined metrics and reproducible evaluation scripts.

2 Background and Related Work

The idea that language models can use tools dates back to early work on tool-augmented generation. Schick et al. [7] introduced Toolformer, showed that the LLMs can learn to insert API calls into their generated text to improve factual accuracy and computation. The key insight was that the model itself can decide when a tool call is needed, rather than relying on hard-coded rules.

More recently, the release of function calling capabilities in commercial APIs like OpenAI's Chat Completions API [8] formalized this pattern. In this system, the developer can describe the available tools as part of the prompt or API schema, and the model returns structured JSON indicating which function to call and with what arguments.

The developer's code then executes the function and feeds the result back to the model for further reasoning. This has become the standard pattern for building LLM agents.

Open-source models have followed a similar trajectory, though with varying levels of support for structured tool calling. Models from the LLaMA family [2], Mistral [3], and Qwen [4] are increasingly fine-tuned or instruction-tuned to produce tool-call outputs, although reliability of the output varies significantly across model families and sizes. Smaller models with less parameters tends to be less consistent in producing correctly formatted tool calls.

2.1 Related work

There are several works which are actively exploring the agents. Such as :

Patil et al. [9] introduced the Berkeley Function Calling Leaderboard, which benchmarks LLMs specifically on their ability to produce correct function calls. Their work provides standardized categories of function-calling difficulty (simple, parallel, multiple, relevance detection) and evaluates both open-source and proprietary models.

Qin et al. [10] proposed ToolLLM, a framework for training and evaluating LLMs on tool use. Their work includes a large-scale dataset of real-world APIs and evaluates pass rate across multi-step tool chains. Xi et al. [11] provide a comprehensive survey on LLM-based agents, categorizing agent designs by their perception, brain, and action modules. Our work fits within their "action" category, specifically addressing the tool-use action module.

3 Use Case and framework overview

3.1 Why agent is needed

A simple question is why AI agent is needed at all. A simple weather application which could call the OpenWeatherMap API directly, format the result, and display it to the user will complete the job. But, A direct API integration would requrire the developer to predict every possible user input and map it to the correct API call via hard coded logic. The user must type a city name in exactly the right format, and the system must have pre-built logic for every possible follow-up action (send SMS, compare cities, ask about tomorrow's forecast, etc.). This works for simple cases but becomes increasingly rigid as the number of supported actions grows.

It would be same as teaching computer to recognize the cat or any other animal. It is a extensive process which could be done in more accurate manner by employing CNNs.

In our project the user can say "What's the weather in Berlin?" or "How's it looking in Berlin today?" or "Tell me if I need an umbrella in Berlin" and the agent can handle all of these because the LLM understands the intent, not just the keywords. Furthermore, when a new tool is added such as a 5 day forecast API or a clothing recommendation function

the agents can integrate it seamlessly. Although, this flexibility comes at a cost , the LLM may misunderstand the user, call the wrong tool, produce malformed arguments, or take unnecessary steps.

So, agents can be game changer in the future development of the software as agents can understand the natural languages, thus give us the entirely new way to interact with the software.

3.2 Framework overview

There are several framework which are in the category of open-source. While new frameworks are also being introduced in the market. The most prominent ones are:

LangChain [5] is arguably the most widely adopted open-source framework for building LLM applications. It provides abstractions for chains (sequential operations), tools (callable functions with schemas), memory (conversation buffers, summaries), and agents (LLMs that choose tools dynamically). LangChain's bindtools method allows attaching tool definitions directly to a model, and the framework handles output parsing, tool dispatch, and result injection into the conversation. LangChain also offers LangGraph [12], a more recent extension for building stateful, graph-based agent workflows.

Microsoft AutoGen [6] takes a different approach, focusing on multi-agent conversation. In AutoGen, agents are distinct entities that communicate through messages. A typical pattern involves a "user proxy" agent that can execute code and a "assistant" agent backed by an LLM. AutoGen excels at collaborative multi-agent scenarios but introduces more complexity for single-agent tool-calling use cases like the one in this project.

4 Implementation

The implementation is built as a tool-using LLM service where natural language interaction is connected to API operations in order to get real time information. The system receives user requests through a FastAPI endpoint and routes those requests through an orchestration layer that decides whether to call external tools. The practical value of the architecture is that it does not stop at language generation. Instead, it retrieves real weather data and can execute function to send SMS to the user

The external service integrations have distinct architectural roles. OpenWeather is used as an external-state provider and is called in two separate functional stages. The first stage resolves a user city name into geographic coordinates through a geocoding API call. The second stage uses those coordinates to query current weather conditions. This split is intentional because weather APIs are coordinate-driven while user prompts are city-driven. Twilio is used as a side-effect service for outbound messaging and is invoked only when workflow policy and user intent allow SMS dispatch. This separation between read-oriented tools and write-oriented tools is important for control, safety, and

observability.

In the service code, these integrations are represented by three tool-level functions: `get_coordinates(city)`, `get_weather(lat, lon)`, and `send_sms(to_number, text)`. Architecturally, `get_coordinates` and `get_weather` form the deterministic weather retrieval chain, while `send_sms` is a policy-sensitive optional step. The orchestration design then determines how and when those tool functions are invoked, and this is where the vanilla and LangChain variants differ.

4.1 Vanilla Architecture

The vanilla architecture uses explicit Python control flow for orchestration. A request enters the API endpoint, session state is resolved, and the model is called with conversation history. The model output is then parsed to identify tool-call intent. If a valid tool-call structure is found, tool functions are invoked in sequence and the resulting observations are injected back into history for subsequent model steps. If tool-call structure is not parseable, the system can still return a conversational response, but deterministic action chaining may be interrupted. This makes the architecture highly transparent and easy to audit, while also making it sensitive to strict output-format assumptions.

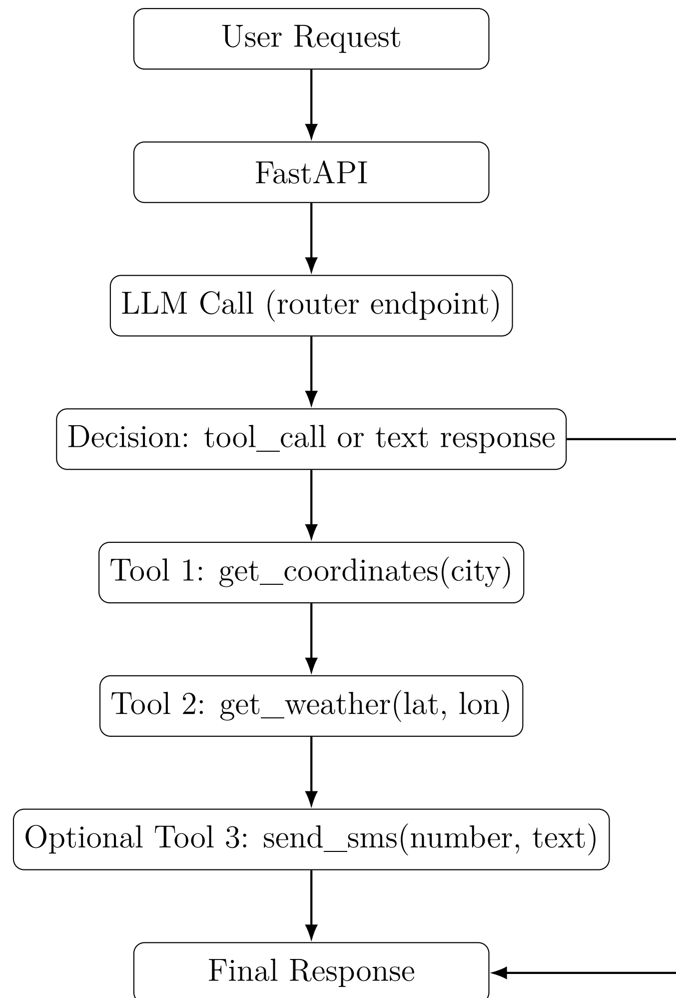

Figure 1: Vanilla orchestration flow with parser-mediated tool chaining.

The architectural strength of this approach is in its predictability as each transition is explicitly coded. The architectural limitation is parsing the LLMs output correctly as the execution depends on strict extraction of tool-call structure from model output.

4.2 LangChain Architecture

The LangChain architecture preserves endpoint behavior but changes orchestration semantics. Instead of free-text parsing for tool intent, tool schema are bound to the model client and tool-call metadata is read from structured model outputs. Execution then follows a bounded step loop where tool calls are dispatched, tool results are returned as typed tool messages, and the model continues until no further tool calls are requested or loop limits are reached. This reduces parser fragility and standardizes tool-execution transitions.

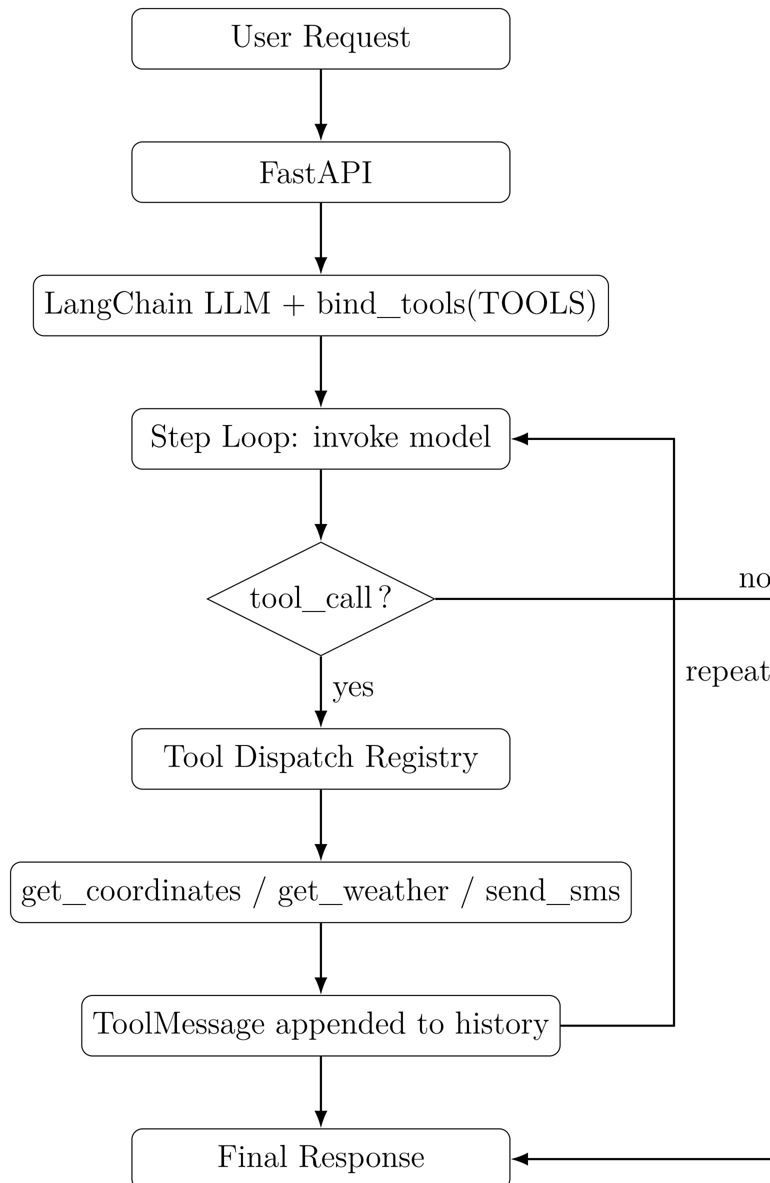

Figure 2: LangChain orchestration flow with structured tool-calling loop.

Compared to the vanilla architecture, the key difference is action-boundary representation. In vanilla flow, action intent is parser-extracted from model text. In LangChain flow, action intent is consumed from structured tool-call metadata. This architectural shift reduces format-dependence and improves consistency of tool execution while keeping the same external product capability.

4.3 Vanilla Hosting Architecture on DigitalOcean Droplet

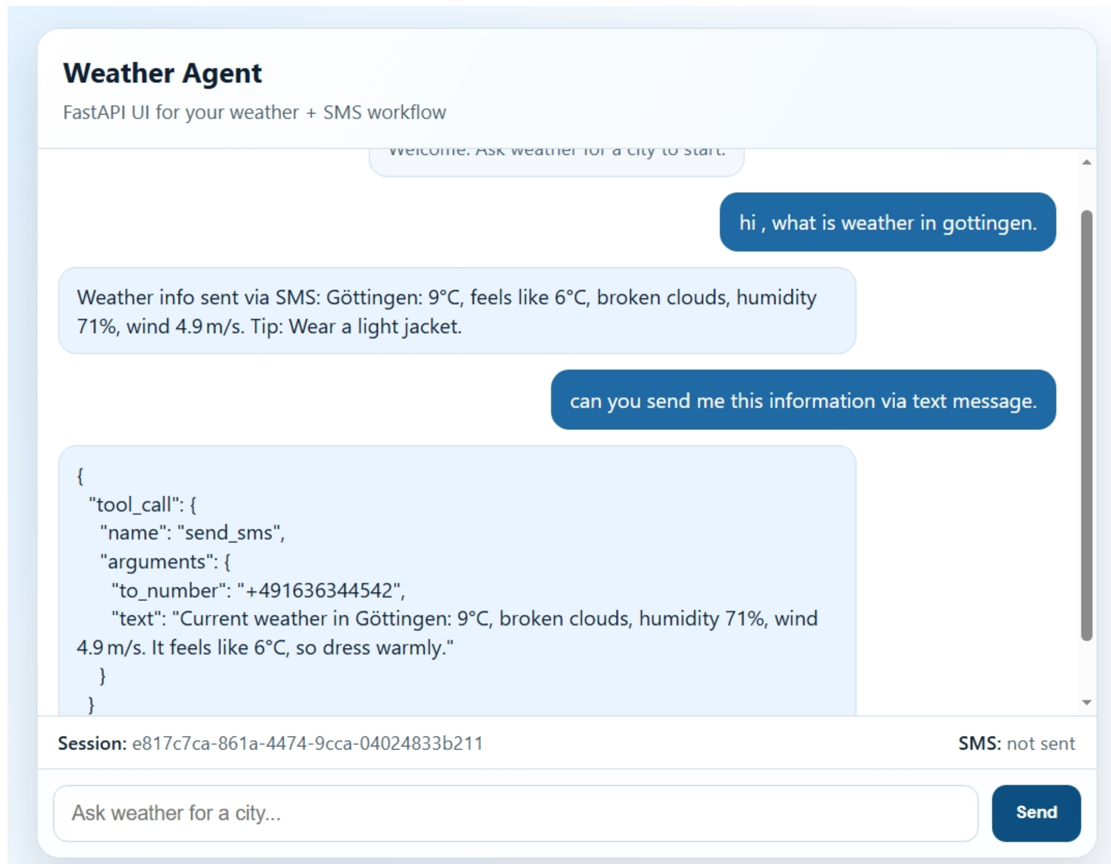

Figure 3: Deployed website interface.

Figure 4: Sample SMS generated by the deployed system.

The hosted implementation of the vanilla system uses a layered service topology. Nginx acts as the public edge and reverse proxy, Uvicorn runs the FastAPI application process, and systemd supervises process life cycle for restart and boot persistence. External integrations remain OpenWeather for weather-state retrieval and Twilio for SMS side effects. This deployment architecture separates edge routing, process hosting, and external API integration into distinct responsibilities, which improves operability and fault isolation. The website can be visited at the address http://209.38.246.241/ .

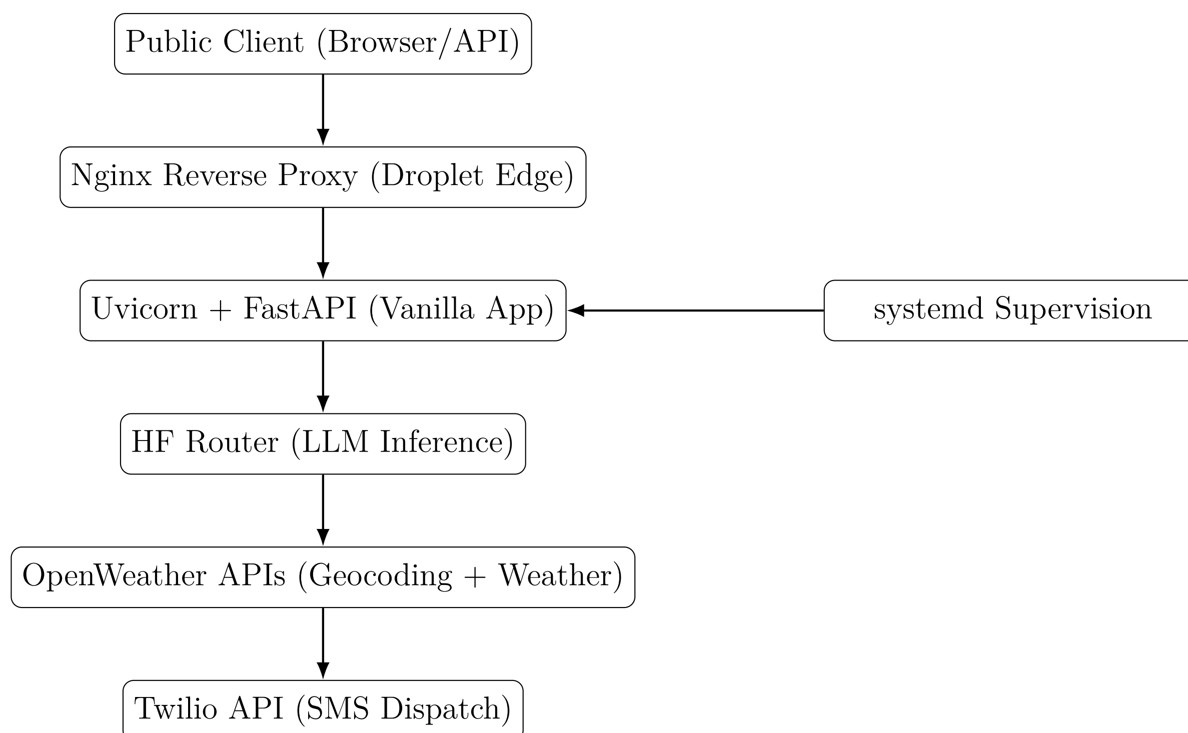

Figure 5: Vanilla deployment architecture on DigitalOcean droplet.

5 Evaluation

This evaluation section measures how reliably and correctly the weather-agent system executes tool-oriented tasks under two orchestration designs: vanilla implementation control flow which means the tool calling and parsing of the LLMs response was done manually without using any frameworks and LangChain structured tool-calling flow.

The goal is not only to compare raw answer quality, but to compare operational behavior under realistic API workloads where correctness depends on tool-path decisions, argument validity, and multi-step completion behavior. Evaluation was executed as repeated HTTP benchmark runs against the same FastAPI interface, while only the orchestration strategy changed.

5.1 Models

The model set was selected to represent practical, widely used, and deployment-relevant instruction models, while still covering meaningful variation in size and capability. The goal was to evaluate both framework behavior and model behavior under the same prompt workload.

1. **Qwen/Qwen3-8B.** This model represents the common 8B class that is popular and can be hosted on the consumer grade hardware. Such, small-to-mid parameter

models are often the first realistic choice in production settings where cost and response time matter.

2. **meta-llama/Llama-3.1-8B-Instruct.** This model was selected as another strong 8B-class baseline from a different model family, allowing cross-family comparison at similar scale. The term Instruct means the model is instruction-tuned: it is additionally trained or aligned to follow user directions in a conversational format, which is important for controlled tool-calling workflows.

3. **openai/gpt-oss-20b.** This model extends the comparison to a larger open model while remaining below frontier-scale size. It was used to test whether moderate parameter growth improves orchestration reliability and argument quality without the full latency cost of very large models.

4. **openai/gpt-oss-120b (baseline reference).** This model was used as a high-capability baseline to compare smaller models against a stronger reference point. Its role is analytical: it helps separate model-capacity effects from orchestration effects (vanilla vs LangChain) and gives an upper-capability anchor for interpretation.

5.2 Parameter used for Evaluation

Five evaluation parameters were used to measure end-to-end reliability, tool-path correctness, and runtime efficiency. Each is computed per run and then aggregated per model.

1. **Completion Rate.** This parameter shows whether runs finish successfully according to the runtime success signal. For each run, completion score is 1 when completion_success is true, otherwise 0. Model-level completion rate is the average of these binary outcomes, so it directly reflects operational reliability.

2. **Tool Accuracy.** This parameter measures positional correctness of the actual tool path against the expected tool path. If expected path length is n, then:

   $$\text{tool_accuracy} = \frac{\#\{i \mid \text{actual}_i = \text{expected}_i\}}{n}$$

   If no tool is expected, the score is 1 only in case when no tool is called.

3. **Argument Accuracy.** This parameter checks whether tool arguments are valid under fixed rules. For get_coordinates, city must be non-empty; for get_weather, latitude and longitude must be in valid numeric ranges; for send_sms, to_number must begin with "+" and text must be non-empty. Per run, argument accuracy is valid argument fields divided by required argument fields.

4. **Step Validity.** This parameter is strict sequence correctness. A run receives 1 only when actual-path exactly equals expected-path; otherwise it receives 0. It is stricter than tool accuracy because partial matches do not receive partial credit.

5. **Mean Latency.** This parameter captures average runtime cost per request in milliseconds. If a model has N runs with latencies $L_1, L_2, \ldots, L_N$, then:

$$\overline{L} = \frac{1}{N}\sum_{i=1}^{N} L_i$$

   Mean latency represents expected response time under repeated use and is useful for comparing practical responsiveness across models.

Evaluation protocol was fixed across both implementations. The benchmark runner iterated through all prompts for each model and recorded one repeat per prompt in this run configuration (EVAL_REPEATS= 1). With 10 prompts per model, each model received 10 benchmark runs per implementation. As in our case, both the vanilla and LangChain implementations were evaluated, each comparable model contributed 20 runs in total across both frameworks. This design provides direct same-prompt comparability between implementations.

5.3 Prompts used for Evaluation

The evaluation suite used 10 prompts, intentionally designed to cover weather retrieval, optional SMS side effects, and non-tool conversational behavior. The exact prompts were:

1. What is the weather in Berlin?
2. Tell me weather in Tokyo and then ask me if I want SMS.
3. Weather in Munich please.
4. Send me weather by SMS for Hamburg.
5. I want weather for Paris and yes send SMS.
6. What is the weather in Delhi now?
7. Hi
8. What can you do?
9. Weather in New York. Do not send SMS.
10. Give weather in Rome and then text it.

These prompts are mapped to expected tool paths so each parameter can be measured in a meaningful way. Prompts 1, 2, 3, 6, and 9 test the standard two-step weather retrieval chain (get_coordinates then get_weather), and therefore stress tool accuracy and step validity for non-SMS weather tasks. Prompts 4, 5, and 10 require the full three-step

chain including send_sms, so they stress multi-step planning, argument correctness for SMS payloads, and completion reliability under side-effect operations. Prompts 7 and 8 are non-tool prompts with empty expected paths, so they verify whether models avoid unnecessary tool calls and preserve conversational behavior.

Because each prompt was run once per model in each implementation, each prompt contributes equally to model scores. This keeps the benchmark balanced and prevents a single prompt type from dominating aggregate metrics.

5.4 Results for Vanilla implementation

Table 1 reports vanilla results for comparable models only (Mistral removed due to provider incompatibility).

Table 1: Vanilla implementation results (comparable models)

Model	Runs	Completion	Tool Acc.	Arg Acc.	Step Validity	Mean Latency (ms)
Qwen/Qwen3-8B	10	1.0000	0.2000	0.2000	0.2000	3107.81
meta-llama/Llama-3.1-8B-Instruct	10	0.8000	0.6667	0.7000	0.4000	3304.75
openai/gpt-oss-20b	10	0.3000	0.5000	0.8000	0.2000	2372.05
openai/gpt-oss-120b	10	0.4000	0.9333	1.0000	0.9000	4202.15

The vanilla implementation shows strong variability across models. The best model in strict orchestration quality is openai/gpt-oss-120b, with tool accuracy 0.9333, argument accuracy 1.0000, and step validity 0.9000. This indicates that when tool chaining succeeds, the model follows the expected sequence with high precision. However, its completion rate is only 0.4000, which indicates that strict end-state completion criteria and multi-step failure points still caused many run-level failures.

Qwen/Qwen3-8B shows an opposite profile: completion rate is 1.0000, but tool accuracy, argument accuracy, and step validity are each only 0.2000. Logs show frequent behavior where the model generated tool-intent-like content in text form that did not reliably convert into parser-accepted tool execution steps. This illustrates a key vanilla limitation: parser coupling can turn seemingly correct model intent into orchestration failure.

meta-llama/Llama-3.1-8B-Instruct provides middle-ground performance. Tool and argument metrics are moderate (0.6667 and 0.7000), but step validity is lower at 0.4000, showing occasional extra, missing, or misordered operations. openai/gpt-oss-20b performs weaker in completion (0.3000), with moderate tool accuracy (0.5000) and better argument accuracy (0.8000), suggesting that argument formation may be acceptable when calls occur, but multi-step stability remains limited.

In latency terms, openai/gpt-oss-20b is fastest in mean latency among comparable vanilla models (2372.05 ms), while openai/gpt-oss-120b is slowest (4202.15 ms), reflecting the expected cost of larger model capacity.

5.5 Results for LangChain implementation

Table 2 reports the LangChain evaluation for the same comparable model set.

Table 2: LangChain implementation results (comparable models)

Model	Runs	Completion	Tool Acc.	Arg Acc.	Step Validity	Mean Latency (ms)
Qwen/Qwen3-8B	10	1.0000	1.0000	1.0000	1.0000	6289.00
meta-llama/Llama-3.1-8B-Instruct	10	1.0000	0.7333	0.8000	0.6000	1406.17
openai/gpt-oss-20b	10	1.0000	1.0000	0.9400	1.0000	1894.26
openai/gpt-oss-120b	10	0.9000	0.9500	0.9400	0.9000	2135.91

The LangChain implementation improves reliability and path correctness for most models. Qwen/Qwen3-8B becomes fully correct across all orchestration parameters (1.0000 on completion, tool accuracy, argument accuracy, and step validity), which is a major improvement over its vanilla behavior. This strongly indicates that structured tool-call handling resolves parser-fragility problems that affected this model in vanilla mode. The trade-off is speed: Qwen has the highest mean latency at 6289.00 ms.

openai/gpt-oss-20b is one of the most balanced LangChain performers, reaching 1.0000 completion, 1.0000 tool accuracy, and 1.0000 step validity, with strong argument accuracy (0.9400) and much lower mean latency than Qwen. In practical terms, this makes it a strong candidate when both correctness and responsiveness are required. openai/gpt-oss-120b remains strong (0.9000 completion, 0.9500 tool accuracy, 0.9000 step validity) with better speed than its vanilla counterpart, although one tool-use parsing failure in logs reduced reliability below perfect.

meta-llama/Llama-3.1-8B-Instruct improves relative to vanilla in completion and speed, but remains less consistent than the best two models in strict sequence control (step validity 0.6000). This reflects residual variability in how the model emits structured calls under some prompts.

5.6 Evaluation Challenges (reproducibility, correctness verification)

A first challenge is provider compatibility and routing stability. One model (Mistral-7B-Instruct-v0.2) repeatedly failed with model-not-supported errors, which means benchmark outcomes can be affected by provider configuration.

A second challenge is the completion-flag interpretation. Completion rate depends on trace-level success flags, which can mark a run incomplete even if major parts of the tool path were executed correctly. This can produce tension between practical usefulness and strict benchmark scoring.

The third challenge is latency between prompts. Some models, especially Qwen in LangChain mode, were consistently slower in average runtime, which affects practical

responsiveness under repeated usage. As these models, were hosted used from the hugging face APIs which means these models were hosted by different inference providers.

A fifth challenge is single-repeat sensitivity. In this run configuration each prompt was evaluated once per model per implementation. While this is useful for controlled comparison, it remains sensitive to run-to-run stochasticity and transient API conditions. Increasing repeats would improve confidence intervals and reduce single-run noise.

6 Discussion

The combined results show a clear architecture effect: LangChain structured tool-calling produces materially stronger orchestration outcomes than vanilla parser-mediated orchestration for most comparable models. A useful way to summarize this is model-average performance across the four comparable models. In vanilla, average completion is 62.50%, average tool accuracy is 57.50%, average argument accuracy is 67.50 %, and average step validity is 42.50%. In LangChain, these become 97.50%, 92.08%, 92%, and 87.50% respectively. In other words, completion improves by 35% and step validity improves by 45% at the framework level.

When we compare model by model, the gains are also visible in concrete values. Qwen/Qwen3-8B moves from 20% tool accuracy and 20% step validity in vanilla to 100% and 100% in LangChain, although mean latency rises from 3107.81 ms to 6289.00 ms. openai/gpt-oss-20b moves from 30% completion and 20% step validity in vanilla to 100% and 100% in LangChain, while also reducing mean latency from 2372.05 ms to 1894.26 ms. openai/gpt-oss-120b improves completion from 40% to 90% and keeps strong path correctness (tool accuracy 93.33% in vanilla versus 95% in LangChain), while mean latency drops from 4202.15 ms to 2135.91 ms. meta-llama/Llama-3.1-8B-Instruct improves completion from 80% to 100% and step validity from 40% to 60%, and its mean latency decreases from 3304.75 ms to 1406.17 ms.

These numbers show that stronger model capacity alone does not automatically guarantee stable orchestration in parser-heavy pipelines. For example, openai/gpt-oss-120b already has high vanilla tool quality (93.33% tool accuracy, 90% step validity), yet completion remains only 40% until the orchestration strategy changes. Conversely, Qwen demonstrates that framework constraints can unlock model behavior, jumping from 20% step validity in vanilla to 100% in LangChain.

The speed-accuracy trade-off is therefore model-dependent rather than uniform. Qwen achieves the highest correctness but at 6289.00 ms mean latency, while openai/gpt-oss-20b reaches near-equivalent correctness (100% in completion,tool accuracy ,step validity and 94% argument accuracy) at 1894.26 ms mean latency. This spread is large enough to matter in deployment decisions, especially when responsiveness and correctness must both be preserved.

The difference between implementations can be explained by action-boundary repre-

sentation. In vanilla mode, action intent must survive free-text generation and parser extraction. In LangChain mode, action intent is represented as structured tool metadata, reducing ambiguity at the orchestration boundary. Because the benchmark tasks are tool-path sensitive, this representation difference directly affects measured correctness.

7 Conclusion

It is evident from the benchmarks across the categories that orchestration design is the primary factor in the reliable tool usage via agent. Across the comparable set,LangChain consistently improved completion reliability and strict path correctness over the implementation done by without using framework.

The most practically balanced performer in this benchmark is openai/gpt-oss-20b under LangChain, while Qwen/Qwen3-8B under LangChain demonstrates the highest achievable correctness at the cost of higher latency. openai/gpt-oss-120b remains a valuable capability baseline and confirms that larger models can provide strong tool discipline, but framework robustness is still required to convert that capability into stable end-to-end execution.

But it also depends upon the developer who is doing the vanilla implementation. A new user or beginner in the field of software development is not expected to cover all the test cases and nuances of the parsing requirements compared to someone with years of experience. In that case, the results will vary and vanilla method may come out on the top. But, overall by using frameworks it is possible to increase the accuracy of the tool calling.

Furthermore, in order to make the evaluation more robust, the number of prompts and the repeat count of the single batch of prompts should increase. This was not done during the report due to APIs limitations.

In the future, with the new framework like Openclaw which give the access to large number of application furthermore increase the score of AI agents. And along with that new open source models which offers better and cheaper performance promise an exciting future for AI agents.

After this project, I plan to implement Openclaw on the server which I am currently renting in order to host the my website. And furthermore evaluating the performance of new open source models and framework with more real world daily use application like sending emails or scheduling appointments.

References

[1] Lei Wang et al. "A Survey on Large Language Model based Autonomous Agents". In: *arXiv preprint arXiv:2308.11432* (2023). arXiv: `2308.11432 [cs.AI]`.

[2] Hugo Touvron et al. "LLaMA: Open and Efficient Foundation Language Models". In: *arXiv preprint arXiv:2302.13971* (2023). arXiv: `2302.13971 [cs.CL]`.

[3] Albert Q. Jiang et al. "Mistral 7B". In: *arXiv preprint arXiv:2310.06825* (2023). arXiv: `2310.06825 [cs.CL]`.

[4] Jinze Bai et al. "Qwen Technical Report". In: *arXiv preprint arXiv:2309.16609* (2023). arXiv: `2309.16609 [cs.CL]`.

[5] LangChain Authors. *LangChain*. GitHub repository. 2022. URL: `https://github.com/langchain-ai/langchain`.

[6] Microsoft. *AutoGen*. GitHub repository. 2023. URL: `https://github.com/microsoft/autogen`.

[7] Timo Schick et al. "Toolformer: Language Models Can Teach Themselves to Use Tools". In: *Advances in Neural Information Processing Systems*. Vol. 36. 2024.

[8] OpenAI. *Function Calling and Other API Updates*. OpenAI Blog. June 2023. URL: `https://openai.com/blog/function-calling-and-other-api-updates` (visited on 03/21/2026).

[9] Shishir Patil et al. "Gorilla: Large Language Model Connected with Massive APIs". In: *arXiv preprint arXiv:2305.15334* (2023). arXiv: `2305.15334 [cs.CL]`.

[10] Yujia Qin et al. "ToolLLM: Facilitating Large Language Models to Master 16000+ Real-world APIs". In: *arXiv preprint arXiv:2307.16789* (2023). arXiv: `2307.16789 [cs.CL]`.

[11] Zhiheng Xi et al. "The Rise and Potential of Large Language Model Based Agents: A Survey". In: *arXiv preprint arXiv:2309.07864* (2023). arXiv: `2309.07864 [cs.AI]`.

[12] LangChain Authors. *LangGraph*. GitHub repository. 2024. URL: `https://github.com/langchain-ai/langgraph`.

A Work Sharing

This report was completed individually.

B Additional Notes

All benchmark tables and architectural diagrams in the main content correspond to the implementation variants described in this report.