Praktikumsbericht

≫ Software für Tool Support in LLMs ≪

Jake :3
E-Mail: jake@stud.uni-goettingen.de

Contents

1. Introduction ... 3
2. Background ... 3
 2.1. WASM ... 3
 2.2. MCP ... 3
 2.3. LLM ... 3
 2.4. OpenAI API ... 4
 2.5. SAIA ... 4
 2.6. Chat AI ... 4
3. Methodology ... 4
 3.1. Architecture ... 4
 3.2. Communication protocol ... 6
 3.3. Sandboxing ... 7
 3.4. Tool server ... 7
 3.5. Tool definitions ... 8
 3.6. Security Threat Model ... 9
  3.6.1. **S**poofing Identity ... 9
  3.6.2. **T**ampering with data ... 10
  3.6.3. **R**epudiation ... 10
  3.6.4. **I**nformation Disclosure ... 10
  3.6.5. **D**enial of Service ... 10
  3.6.6. **E**levation of Privilege ... 11
 3.7. Deployment Model ... 11
 3.8. Benchmarks ... 11
4. Evaluation ... 12
 4.1. Performance ... 12
5. Discussion ... 13
 5.1. MCP ... 13
 5.2. WASM ... 13
 5.3. Python Tools ... 14
6. Conclusion ... 14
Bibliography ... 15
A Tables ... 17
B Screenshots ... 18

Declaration on the use of LLMs in the context of examinations

In this work I have used LLMs as follows:

- ☐ Not at all
- ☒ During brainstorming
- ☐ When creating the outline
- ☒ To write individual passages, altogether to the extent of writing Section 6.
- ☐ For the development of software source texts
- ☐ For optimizing or restructuring software source texts
- ☐ For proofreading or optimizing
- ☒ Further, namely to write code to visualize benchmarking results.

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

1. Introduction

Large Language Models (LLMs) are able to understand imprecise human language and are able to generate statistically likely responses, yet they still struggle with precise tasks such as counting and math. To improve on these shortcomings the ability for the LLM to outsource these more precision demanding tasks onto an external service is required. This external service should provide tools for the LLM to call when required. Tool capabilities should range from purpose build for specific tasks to running arbitrary code in common languages such as Python. When a tool is called it should be able to be run securely sandboxed on HPC infrastructure.

In this work we design, implement, and evaluate a tool server which implements such a service and discuss how to integrate it into existing infrastructure.

2. Background

This section introduces concepts that are required as background information for this report.

2.1. WASM

WebAssembly (WASM) is a standarized portable binary assembly format designed for running sandboxed compiled functions inside of a virtual machine in a browser with minimal overhead. Many modern languages, such as C, Python, Rust, and Go, have compilers and tools that allow targeting WebAssembly as a compilation target. A programmer can select and run specific functions of a WASM binary through a WASM runtime. [1]

WebAssembly is designed to be secure by default. A WASM function inside of a WASM binary may only access host functionality via functions explicitly provided by the runtime. [1]

The WebAssembly System Interface (WASI) is a group of API specifications designed to compile and run generic applications for WASM that require POSIX-like syscalls. [2] WASI defines functions which an application may use to emulate OS-like functionality such as filesystem access, the current time, and random number generation.

2.2. MCP

Model Context Protocol (MCP) is a protocol introduced by Anthropic in 2024 designed for providing Resources and Tools (called *Servers*) to LLMs via a so called *Client*. [3] MCP messages are encoded using JSON-RPC and can be transferred between client and server through stdio or over Streamable HTTP. [4] There are official SDKs available for multiple common languages including Python, Java, and TypeScript as well as community maintained SDKs for languages such as Go. [5], [6]

The LLM gets a definition of all available tools in the system prompt and may choose to call any of them by generating an appropriate response. [7]

2.3. LLM

Large Language Models (LLM) are transformer models which are pretrained on large amounts of data to predict the statistically likely next word in a given sequence words. [8] By repeatedly predicting the next word entire sentences and paragraphs can be generated.

2.4. OpenAI API

The OpenAI API is reference for an API for interacting with LLMs over HTTP. [9]

The API supports tool calling by giving a machine readable description and usage information of available tools to the LLM on a request. The response may then contain a machine readable request to call a given tool. [10] The client can then interpret that tool call request and perform the tool call.

2.5. SAIA

Scalable AI Accelerator (SAIA) is a solution for running LLMs on HPC infrastructure. It includes services such as Chat AI. [11]

2.6. Chat AI

Chat AI is a web service for interacting with LLMs hosted by the KI-Servicezentrum für Sensible und Kritische Infrastrukturen (KISSKI) and the Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen.

The web frontend uses React, is open source and developed in-house.

3. Methodology

In the following sections we design, discuss and document the requirements and implementation details of the tool server, how the tool server should be integrated into existing architecture and how it was benchmarked.

3.1. Architecture

The first thing to consider for designing the tool server is where it will fit into the existing architecture of the Chat AI environment as seen in Figure 1.

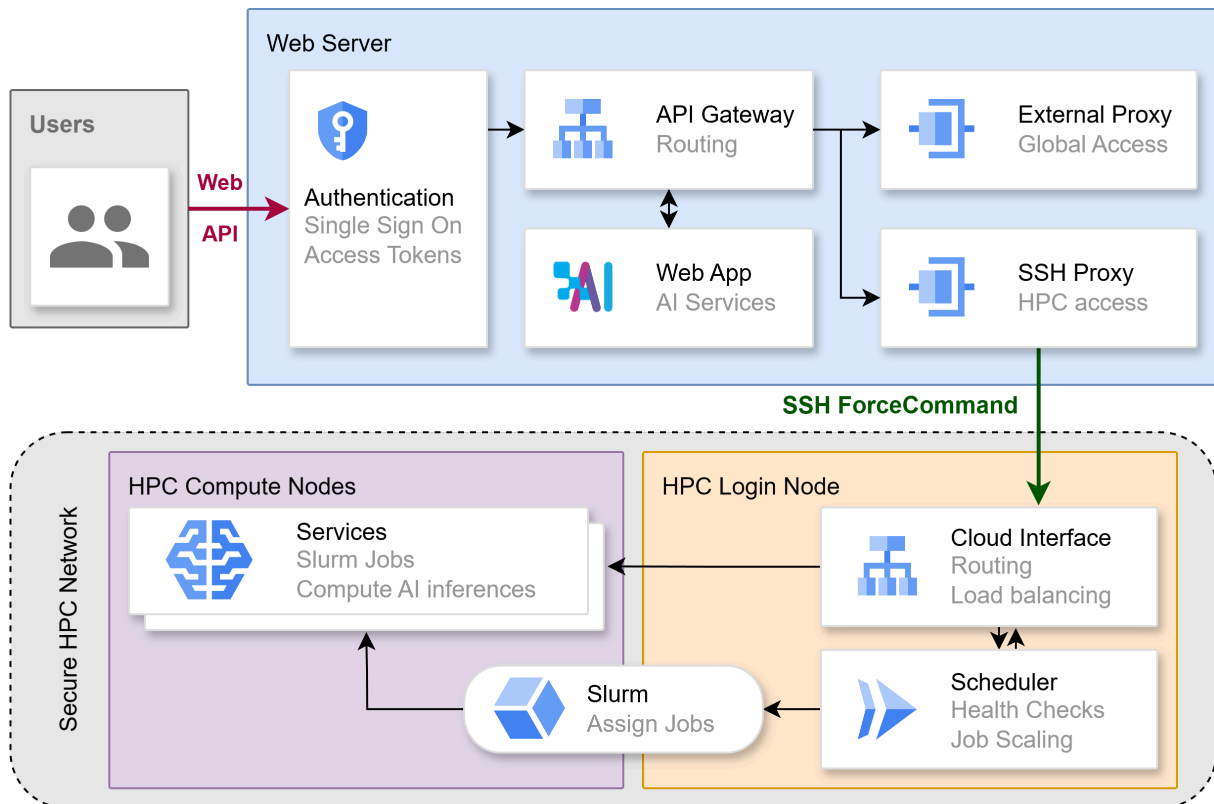

Figure 1: Chat AI Architecture diagram [12]

One could implement the tool server as a vLLM middleware which would sit on the servers running the LLMs. This approach has multiple drawbacks, such as but not limited to

- the user doesn't have control over which tools are called,
- the system is less flexible, any deployment of new tools would necessitate redeploying all vLLM instances,
- supporting tool calls for external LLMs (e.g. Open AI models) would be impossible, and
- the GPU cluster machines running the vLLM software should not be burdened by unnecessary extra load.

A few of these drawbacks could be addressed by implementing the tool server not as a vLLM middleware, but as a middleware for the saia hub. This would allow for adding support for tool calls on external LLMs and would also remove the burden on the vLLM machines. To deploy one would also only need to redeploy the saia hub software. Only the issue of user control still remains.

By implementing the tool server as a standalone service with direct access by the user and implementing the tool calling inside of the Chat AI frontend we can address all of the stated issues at once.

- The user has full control over which tools are available to the LLM. Even editing of arguments by the user before a tool call is executed is possible.
- Deploying new tools just requires updating the tool server itself.
- Tool callings would also work for external LLMs, as the frontend treats external LLMs and internal LLMs the same.
- The GPU cluster machines running vLLM do not get any extra load.

Furthermore using this approach allows the user to provide their own tools and tool servers. Providing tools that will run inside the browser itself is also possible using this approach.

In the end it was decided to use the standalone service approach with tool calling being implemented inside the Chat AI frontend as this approach has the most benefits.

There are three main questions for designing the tool server:
1. How can the LLM know which tools are available?
2. How can the LLM call a tool on the tool server?
3. How can the tool server run the tools?

3.2. Communication protocol

With the given architecture there are now the following problems that need to be solved for communicating between the LLM and the tool server.

First the Chat AI frontend needs to ask the tool server which tools are available, which it then needs to forward to the LLM in a format it can understand. After that the LLM might want to call a tool, this needs to be caught inside the frontend, translated into a format understandable for the tool server and forwarded to the tool server for execution. The result from the tool execution then needs to be converted back into a format the LLM can understand, added to the conversation and be send back to the LLM so it can generate its final response. The final response can then be added back to the conversation too. An overview of these steps can be seen in Figure 2.

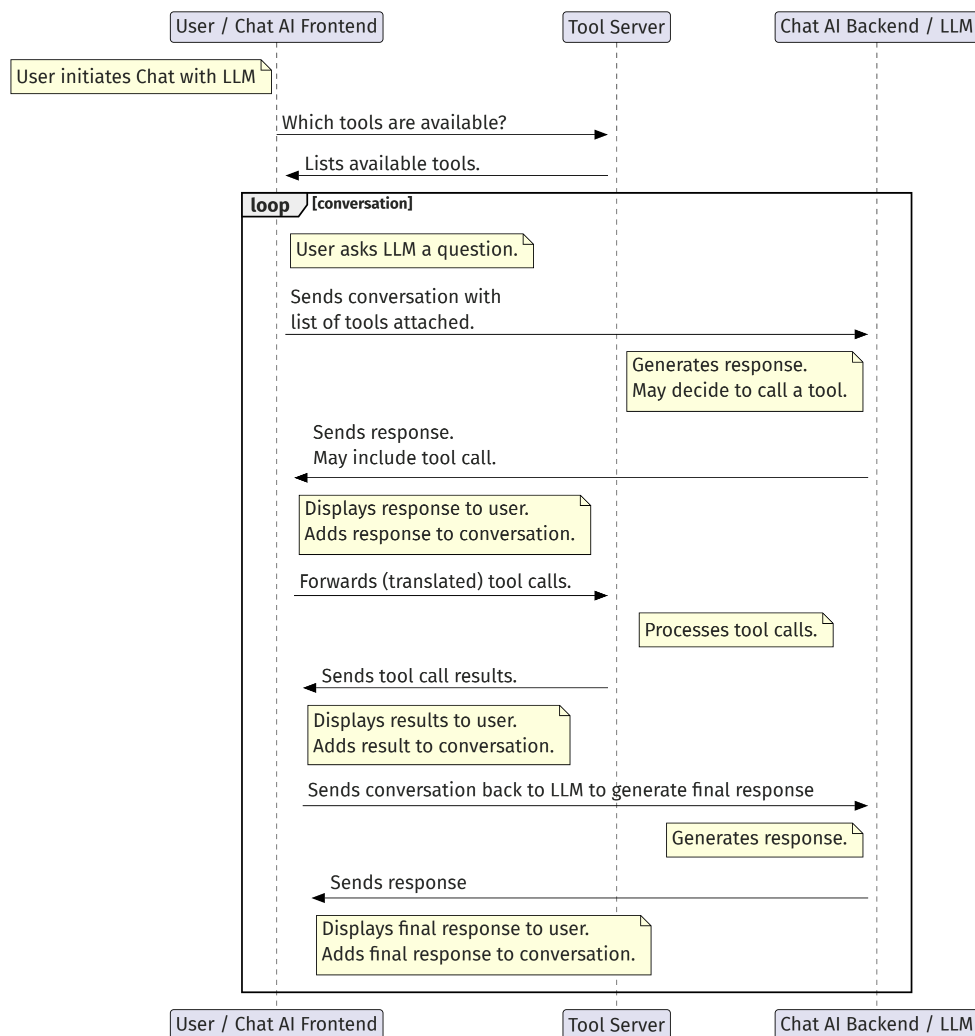

Figure 2: Overview of communication protocol.

As mentioned in Section 2.4 the OpenAI API supports tool calls where the LLM gets a list of available tools and then contains an array of machine readable tool calls which the LLM wants to perform in the response. Thus we already have the means for communicating the tool calls with the LLM.

Now we only need a protocol for communicating with the tool server itself. For this we could implement our own simple protocol. This could work for example as an API provided over HTTP. There could be an endpoint for listing the available tools and an endpoint for calling a given tool.

However, it is usually better to use industrywide established standards. One such defacto standard is the MCP protocol as described in Section 2.2. Using this protocol for the communication allows us to implement the tool server as a generic MCP server and enables us to use any existing MCP server not only our tool server. The Chat AI frontend only needs to implement an MCP client which can easily be done using the official TypeScript SDK. [13] There are also existing debugging tools for the MCP protocol, thus enabling us to debug the tool server itself independently from the Chat AI frontend. [14]

Given these reasons it was decided to use the MCP protocol as the communication protocol.

The main problem with using the MCP protocol are the differences in the OpenAI format and the MCP format for tool definitions and tool calls. The Chat AI frontend needs to be able to translate

- the tool definitions given by the MCP server into OpenAIs format,
- the tool calls from the LLM response over the OpenAI API into the MCP format, and
- the result of the tool call back into the OpenAI format for inclusion in the conversation.

The translation is for the most part rather simple as both formats try to accomplish similar goals.

3.3. Sandboxing

We need to sandbox tool execution to ensure malicious users cannot abuse our systems or gain access to other users' data.

Our sandboxing approach should satisfy the following constraints:

- It should be fast to start.
- Sandboxed environments should be fully independent from each other and thus be able to work in parallel.
- The sandbox should be secure by default. If we forget to configure, for example networking, then it should fall back to a secure default.
- The performance overhead should be minimal.
- Resources should be limited.

Sandboxing of an application can be achieved through virtualization or containerization. [15]

3.4. Tool server

Given that we decided to use the MCP protocol we can now think about the tool server implementation.

The tool server software has multiple requirements.

1. The server needs to host an MCP server.
2. Tools need to be defined. Preferably through a configuration file.
3. The server needs to be able to execute a given tool.

The tool server is implemented in Go using **the mcp-go library**[1] .

There are multiple ways one could implement tools and tool execution.

The first naive approach would be to simply implement the tools as normal Go functions. This would limit tools to being written in Go and would require a complete rebuild of the tool server for every change.

The second naive approach would be to simply run commands. This approach allows already existing programs to be used as tools. Whenever someone would for example want to run some python code, then the tool server would simply spawn a shell and run the OS native `python3` executable. While this would allow tools to be basically written in any language, it would also make it very hard to properly sandbox the tools and thus increase the risk of security problems.

To address these problems we need a generic way to run arbitrary applications in a secure sandbox. One option that can be considered here would be to define tools as docker images and simply create and run a new container per tool call. However there are multiple drawbacks for this approach too. Sandboxing docker containers correctly with proper isolation and restrictions is not an easy task. [16] Furthermore is starting a docker container rather slow and resource heavy, which is not ideal if done per tool call. [17]

If we want proper isolation and secure sandboxing we can look at virtual machines. Running a complete operating system in a – for example – x86 virtual machine would solve most of our problems stated above but again be too slow and resource heavy to be done on every tool call. Instead of running a complete operating system we can define tools in webassembly as described in Section 2.1.

In the end it was decided to use the **wasmtime webassembly runtime**[2] to run tools.

A tool can be any WASI application.

Any WASI application can now be configured to be a tool in conjunction with the necessary information for MCP and for building the WASI environment: Information required for MCP:
- the name of the tool
- the description of the tool
- the input schema

Information required for WASI environment:
- argv
- environment variables
- templates for files to exist

3.5. Tool definitions

Given a WASM/WASI file, we can now define a tool with the following information:
- The name for the tool in MCP.
- The description for the tool in MCP.
- The input schema for the tool in MCP.
- The path to the WASM file.
- Which files should be made available to the tool inside the WASI runtime and where. These files should contain information from the provided MCP parameters.

[1] https://pkg.go.dev/github.com/mark3labs/mcp-go@v0.31.0
[2] https://wasmtime.dev/

- The environment variables made available to the tool.
- The Argv value with which the tool is supposed to be called.
- Optionally we can set a maximum number of actions the tool can perform called “fuel”, which is roughly equivalent to the number of WebAssembly instructions to be run. [18]
- Optionally we can also set a maximum runtime for the tool with a deadline in seconds.

In Listing 1 is an example tool definition in JSON.

```json
{
  "enabled": true,
  "name": "python3-slim",
  "description": "Run the given python code. No external packages are installed. Any file
generated with the code will automatically be part of the response and shown to the
user.",
  "schema": {
    "type": "object",
    "properties": {
      "code": {
        "type": "string",
        "description": "The python3 code to run."
      }
    },
    "required": [
      "code"
    ]
  },
  "wasm_file": "./tools/out/python-3.12.0.wasm",
  "mnt_dir": "/",
  "files": {
    "code.py": "{{.code}}"
  },
  "argv": [
    "python3",
    "/code.py"
  ],
  "env": {},
  "fuel": 500000000000,
  "fuel_enabled": true,
  "deadline": 120,
  "deadline_enabled": true
}
```

Listing 1: This defines a tool named “python3-slim” which expects a property named “code” in its input. The tool is loaded from the WASI binary located at “./tools/out/python-3.12.0.wasm” on the server. Inside the sandbox environment a file named “code.py” is placed inside the “/” directory and is filled with the contents of the “code” property from the input. Any extra files that will be generated by the tool and placed inside of “/” during execution will be send back to the user as well as the stdout and stderr output of the program. The WASI binary is executed with the arguments “python3 /code.py”. There are no further environment variables set. Resource limits are configured.

3.6. Security Threat Model

Whenever we interact with user data security must be considered as a significant factor. We can use the **STRIDE** threat list to determine potential threats. [19]

3.6.1. Spoofing Identity

A malicious user may try to imitate another user to gain access to that users data. [19]

The tool server is designed to be stateless and has no further authentication nor identification by itself. Every request produces one response with no lingering user data on the server.

Due to this stateless approach spoofing is not possible.

If authentication and access control is desired it should be handled before the user is able to access the tool server for example by limiting access through a reverse proxy with authentication configured and enforced.

3.6.2. Tampering with data

A user may try to maliciously tamper with data of other users either stored on disk or in transit. [19]

As mentioned earlier the tool server is stateless with no lingering user data on the server, thus there is no persistent data that can be tampered with. However during a request user data may be stored on the server and transmitted over the network.

The data stored on the server during a request is protected by each request being handled seperatly inside of the tool server and tool calls being handled in sandboxed environments as discussed in Section 3.4 and Section 3.3.

Protection of user data in transit over the network can be achived by encrypting the communication between user and server. This can be achieved by limiting access to the tool server through a reverse proxy with encryption configured and enforced.

3.6.3. Repudiation

The tool server currently does not perform any logging. Should a user be able to perfom prohibited operations on the server it is very hard to trace their actions.

If access to the tool server is limited through a reverse proxy as described in previous sections then that reverse proxy may be configured to implement some rudimentary logging of metadata. In practice this would probably mostly only enable tracing who was using the server and thus who was potentially affected or responsible, but would likely not enable a full trace of actions.

3.6.4. Information Disclosure

A malicious user may try to read data of other users either stored on disk or in transit. [19]

The same mitigations as discussed in Section 3.6.2 apply.

3.6.5. Denial of Service

A malicious user may try to deny access to the server for other users or make the service unusable through resource exhaustion. [19]

Every tool can be configured with resource limits for number of actions and runtime as described in Section 3.5. Thus a single request cannot be used to exhaust all server computing resources. However there may be currently a denial of service attack possible through exhaustion of memory, though this can of course only be done temporarily if tools have a maximum runtime configured. [20]

The tool server does not implement any form of rate limiting. A user can achieve resource exhaustion by spamming many requests in parallel which try to use as many resources as possible. This can be mitigated by limiting access to the tool server through a proxy server with rate limiting configured and enforced. If the proxy server has authentication configured as described in Section 3.6.1 then the rate limiting may also be configured per user.

3.6.6. Elevation of Privilege

A malicious user may try to gain privileges to access information they are not permitted to see, resources they are not permitted to use, or otherwise try to compromise the system. [19]

These threats are again mitigated through the sandboxing approach as described in Section 3.4 and Section 3.3

As the tool server may be deployed in a secure network behind a reverse proxy we must focus on network security, specifically Server-Side Request Forgery (SSRF). [21] SSRF is mitigated by prohibiting all network access for tools in the sandbox.

3.7. Deployment Model

The tool server can be compiled to a single statically linked executable. A configuration file and the corresponding related tool wasm binaries are required if deploying the tool server manually by simply running the binary.

It is recommended to deploy the tool server instead via the **official docker image**[3] . Further documentation for deploying using the docker image can be found in the **README**[4] .

The tool server does not provide any form of authentication, access control, rate limiting, or logging. Thus the tool server should be deployed behind a reverse proxy which implements these features and only be accessible through this reverse proxy.

3.8. Benchmarks

To evaluate the performance of the tool server, a series of benchmarks were conducted.

Performance benchmarks were conducted by using the command line benchmarking tool **hyperfine**[5] which would use **curl**[6] to automatically send requests to a local instance of the tool server and measure their response times. Three "warmup" requests were sent before taking measurements.

Benchmarks were run in one sitting on an Intel i5-11500T CPU running Ubuntu 24.04.3 LTS. The tool server instance was never stopped during the testing.

We configured four additional tools to the `python3-slim` and `python3-full` tools:
- `benchmark-wat-helloworld`: A "Hello World" program written in pure WebAssembly.
- `benchmark-c-helloworld`: A "Hello World" program written in C.
- `benchmark-c-pi`: A program written in C which calculates π using the Bailey–Borwein–Plouffe formula with a given n. [22]

The C programs were compiled to WebAssembly using the Clang compiler version `18.1.3 (1ubuntu1)` with `-O2`.

For benchmarking the `python3-slim` and `python3-full` tools we similarly ran an empty program, a simple "Hello World" program and a program which calculates π using the Bailey–Borwein–Plouffe formula with a given n.

[3] https://gitlab-ce.gwdg.de/hpc-team/scalable-ai/tool-server/container_registry
[4] https://gitlab-ce.gwdg.de/hpc-team/scalable-ai/tool-server#deploy
[5] https://github.com/sharkdp/hyperfine
[6] https://curl.se/

4. Evaluation

4.1. Performance

The results of the benchmarks as described in Section 3.8 can be found in Table 1 and Figure 3.

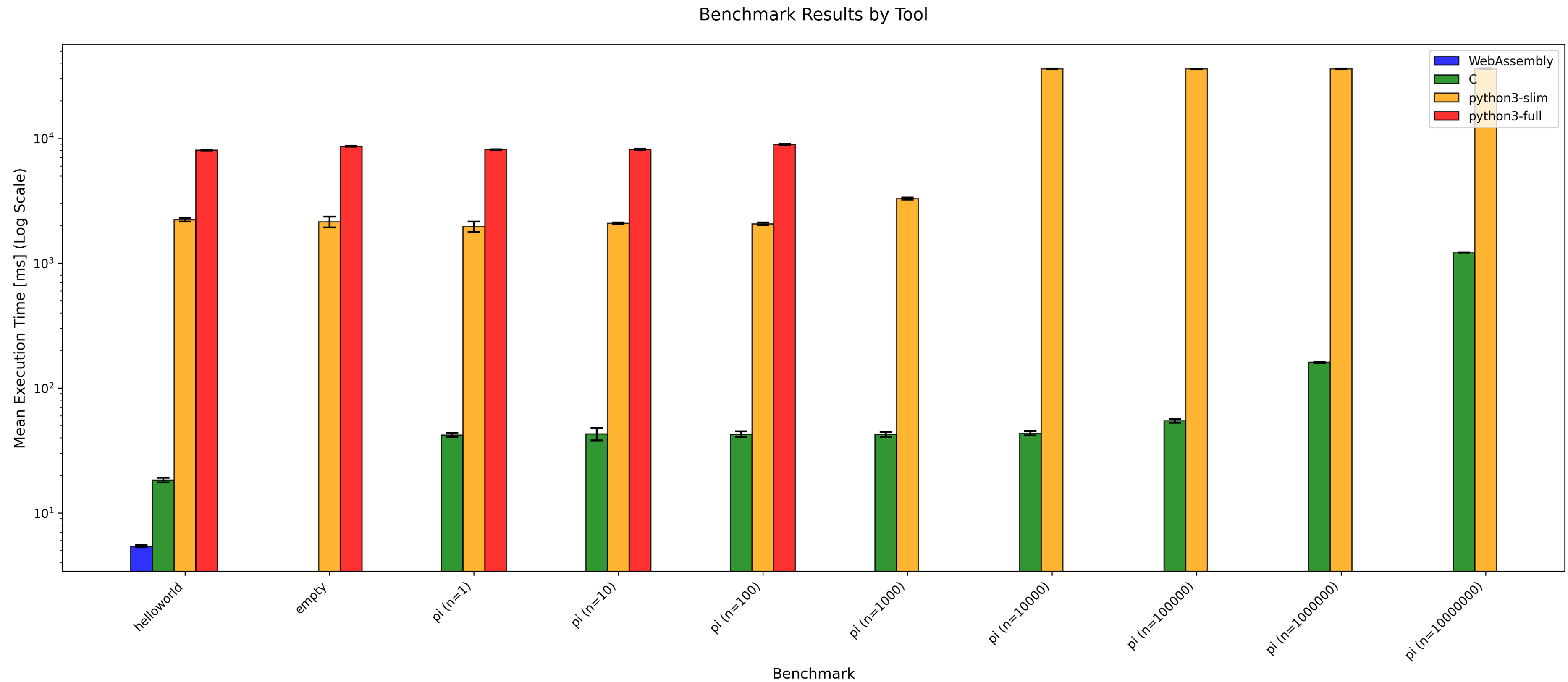

Figure 3: Visualized Benchmark Results of Table 1.

The “Hello World” benchmark written in pure WebAssembly was the fastest with an average execution time of around 5.4 ms. benchmarking was done by measuring the full round trip time of sending a request and receiving the response, thus we can infer that the overhead the tool server introduces for decoding the request, starting the sandbox and sending the response is less than 5.4 ms.

The “Hello World” benchmark written in C is around 10ms slower than the pure WebAssembly one.

The “Hello World” and empty benchmarks written in Python and running on `python3-slim` take over two seconds to run. This is likely the overhead required for starting the python interepreter. Notably the empty benchmark took longer to run than the “Hello World” benchmark indicating a higher margin of error as originally estimated. The π benchmarks from $n = 1$ to $n = 100$ also ran in about two seconds with no notable increase. After that the time increases slightly to three seconds at $n = 1000$ and after that the time remains around 36 seconds.

This behaviour may be explained by the Cranelift just-in-time compilation optimization of Wasmtime. [23] At first the execution time of the loop up to $n = 100$ is insignificant next to the instantiation time. After $n = 1000$ the optimization kicks in and the roughly 34 seconds are used for the compilation. After that the execution time of the loop once again becomes insignificant.

A similar behaviour can be observed with the π benchmarks written in C though there we don’t see the optimization benefits at the end.

All Benchmarks that were successfully run using the `python3-full` tool consistently took roughly eight seconds to run. The tool failed to complete the more complex π benchmarks.

5. Discussion

In the following sections we discuss the limitations and design choices of the tool server and its tools.

5.1. MCP

The usage of MCP for the communication protocol was the correct choice. It is powerful enough to handle everything that we required for the implementation of the tool server and enables the tool server to be easily integrated into existing infrastructure that already supports MCP such as SAIA.

The largest problem are the differences in the formats of tools and tool calls of the OpenAI API and MCP. However it is easily possible to translate between those formats.

An official Go SDK for MCP has been released since the development of the tool server began. [24] While the community build mcp-go library is currently still maintained, usage of the official SDK should be preferred. [25]

5.2. WASM

Using WASM as a sandboxing solution did result in multiple issues.

Tools being stateless and lacking internet access limits their ability for many usecases.

The concept of using WASI applications as tools does in theory result in the ability to easily integrate many existing prebuilt applications as tools. In practice however it is difficult to find prebuilt WASI applications that are useful as stateless tools for the usage with LLMs.

While container2wasm can in theory be used to convert any application into a WASI application, the emulation overhead is too large for it to be useful in practice as can be seen in the benchmark results of the python3-full tool in Section 4.1. container2wasm does also not forward the exit codes properly causing false negatives in the error detection.

5.3. Python Tools

There are currently two configured tools for running Python: python3-slim and python3-full

python3-slim is based on a WASI Python 3.12.0 runtime built using CPython's WASM+WASI support. [26] As seen in Section 4.1 it has an acceptable performance for simple tasks. The tool is not able to use python packages and thus missing crucial features such as plot generation with matplotlib or working with matricies with numpy.

python3-full is build with container2wasm using the official Python 3.13.5 Alpine 3.22 Docker-image. It contains support for the numpy, scipy, pandas, matplotlib, and seaborn packages including all their dependencies. As can be seen in Figure 4 and Figure 5 the performance on simple tasks is not acceptable for production use.

A software like **Pyodide**[7] should be used instead, which does support these packages and can be compiled directly to WebAssembly. Pyodide does not support compiling to WASI, which makes it not usable in the current tool server. [27]

6. Conclusion

In conclusion, we presented the design and implementation of a tool server for Large Language Models that utilizes WebAssembly and the Model Context Protocol to provide a secure and flexible way to execute precision-demanding tasks. The tool server allows users to define and execute tools as WASI applications, which can be run in a sandboxed environment to ensure security and isolation. We discussed the architecture and design of the tool server, including the use of MCP as a communication protocol and the implementation of tool execution using the wasmtime webassembly runtime. However, we also highlighted several issues and limitations with the current design and implementation of the tool server, including the limitations of stateless tools, the difficulty of finding prebuilt WASI applications, and the performance overhead of using container2wasm to convert applications to WASI. Additionally, we noted that the current tools for running Python are either too slow or lack necessary features, and that alternative solutions such as Pyodide may be more suitable. Overall, we demonstrated the potential of using WASM and MCP to provide a secure and flexible way to execute precision-demanding tasks for LLMs, but has also highlighted the need for further development and improvement to address the current limitations and issues. Future work should focus on addressing these limitations and exploring alternative solutions to provide a more robust and efficient tool server for LLMs.

[7] https://pyodide.org/

Bibliography

[1] "WebAssembly." Accessed: Feb. 03, 2026. [Online]. Available: **https://webassembly.org/**

[2] "WebAssembly System Interface (WASI)." Accessed: Feb. 02, 2026. [Online]. Available: **https://wasi.dev/**

[3] "Introducing the Model Context Protocol," Anthropic Announcements. Accessed: Jan. 05, 2026. [Online]. Available: **https://www.anthropic.com/news/model-context-protocol**

[4] Anthropic, "Model Context Protocol Specification - Transport." Accessed: Feb. 23, 2026. [Online]. Available: **https://modelcontextprotocol.io/specification/2025-11-25/basic/transports**

[5] "SDKs," MCP Documentation. Accessed: Feb. 03, 2026. [Online]. Available: **https://modelcontextprotocol.io/docs/sdk**

[6] *mcp-go*. Accessed: Feb. 03, 2026. [Online]. Available: **https://github.com/mark3labs/mcp-go**

[7] Anthropic, "Model Context Protocol Specification." Accessed: Jan. 05, 2026. [Online]. Available: **https://modelcontextprotocol.io/specification/2025-11-25**

[8] Cole Stryker, "What are LLMs?," The 2026 Guide to Machine Learning. Accessed: Feb. 03, 2026. [Online]. Available: **https://www.ibm.com/think/topics/large-language-models**

[9] "API Reference Introduction," OpenAI API Reference. Accessed: Feb. 03, 2026. [Online]. Available: **https://platform.openai.com/docs/api-reference/introduction**

[10] "Function calling," OpenAI Platform. Accessed: Feb. 03, 2026. [Online]. Available: **https://platform.openai.com/docs/guides/function-calling**

[11] A. Doosthosseini, J. Decker, H. Nolte, and J. Kunkel, "SAIA: A Seamless Slurm-Native Solution for HPC-Based Services." p. , 2025. doi: **10.21203/rs.3.rs-6648693/v1**[8] .

[12] "Chat AI Architecture Diagram," Chat AI. Accessed: Feb. 02, 2026. [Online]. Available: **https://github.com/gwdg/chat-ai/blob/5b5355f449bffc2c08652a9b57f181e868e802d0/assets/arch-diagram.png**

[13] *MCP TypeScript SDK*. Accessed: Feb. 02, 2026. [Online]. Available: **https://github.com/modelcontextprotocol/typescript-sdk**

[14] "Debugging," MCP Documentation. Accessed: Feb. 02, 2026. [Online]. Available: **https://modelcontextprotocol.io/legacy/tools/debugging**

[15] Ed Moyle and Matthew Pascucci, "What's the difference between sandboxes vs. containers?," TechTarget. Accessed: Feb. 02, 2026. [Online]. Available: **https://www.techtarget.com/searchsecurity/answer/Whats-the-difference-between-software-containers-and-sandboxing**

[16] Matt Conran: The Visual Age, "Docker Container Security," Network Insight. Accessed: Feb. 02, 2026. [Online]. Available: **https://network-insight.net/2022/07/07/docker-container-security-building-a-sandbox/**

[8] https://doi.org/10.21203/rs.3.rs-6648693/v1

[17] Shamsher Khan, “Understanding Docker Startup Performance: A Three-Tier Analysis From 303ms to 837ms,” OpsCart. Accessed: Feb. 02, 2026. [Online]. Available: **https://opscart.com/understanding-docker-startup-performance/**

[18] “Wasmtime Store set_fuel() function,” Wasmtime Documentation. Accessed: Jan. 05, 2026. [Online]. Available: **https://docs.wasmtime.dev/api/wasmtime/struct.Store.html#method.set_fuel**

[19] “Threat Modeling Process,” OWASP Community Pages. Accessed: Jan. 05, 2026. [Online]. Available: **https://owasp.org/www-community/Threat_Modeling_Process**

[20] Marco @zhenjunMa, “How to set resource limit in wasmtime-go?.” Accessed: Feb. 02, 2026. [Online]. Available: **https://github.com/bytecodealliance/wasmtime-go/issues/101**

[21] “Server-Side Request Forgery Prevention Cheat Sheet¶,” OWASP Cheat Sheet Series. Accessed: Feb. 03, 2026. [Online]. Available: **https://cheatsheetseries.owasp.org/cheatsheets/Server_Side_Request_Forgery_Prevention_Cheat_Sheet.html**

[22] D. Bailey, P. Borwein, and S. Plouffe, “On the Rapid Computation of Various Polylogarithmic Constants,” *Mathematics of Computation*, vol. 66, no. 218, pp. 903–913, 1997, doi: **10.1090/S0025-5718-97-00856-9**[9] .

[23] *Wasmtime*. Accessed: Feb. 02, 2026. [Online]. Available: **https://github.com/bytecodealliance/wasmtime**

[24] *MCP Go SDK*. Accessed: Feb. 03, 2026. [Online]. Available: **https://github.com/modelcontextprotocol/go-sdk**

[25] Ed Zynda @ezynda3, “Why do you still use mcp-go when an official Go SDK exists?.” Accessed: Feb. 03, 2026. [Online]. Available: **https://github.com/mark3labs/mcp-go/discussions/662**

[26] “Python WebAssembly Language Runtime.” Accessed: Feb. 03, 2026. [Online]. Available: **https://github.com/webassemblylabs/webassembly-language-runtimes/tree/main/python**

[27] “compile wasi version wasm file.” Accessed: Feb. 03, 2026. [Online]. Available: **https://github.com/pyodide/pyodide/issues/3535**

[9] https://doi.org/10.1090/S0025-5718-97-00856-9

A Tables

Benchmark	n	Lang/Tool	Mean [ms]	Min [ms]	Max [ms]
`helloworld`		WebAssembly	5.4 ± 0.1	5.1	6.0
		C	18.3 ± 0.8	17.5	24.1
		`python3-slim`	2225.0 ± 73.7	2106.2	2310.0
		`python3-full`	8038.2 ± 45.3	7928.1	8102.3
`empty`		`python3-slim`	2145.0 ± 208.6	1805.5	2339.7
		`python3-full`	8615.6 ± 67.9	8484.2	8700.6
`pi`	1	C	42.0 ± 1.5	40.4	51.5
		`python3-slim`	1961.5 ± 185.4	1601.6	2114.5
		`python3-full`	8091.0 ± 78.0	7.952	8.239
	10	C	42.9 ± 4.9	40.4	78.4
		`python3-slim`	2078.9 ± 34.0	2018.2	2142.9
		`python3-full`	8136.0 ± 88.0	7.978	8.318
	100	C	42.7 ± 2.2	40.5	51.9
		`python3-slim`	2065.8 ± 46.3	1985.4	2130.9
		`python3-full`	8917.0 ± 58.0	8.819	9.015
	1000	C	42.5 ± 2.0	40.8	51.4
		`python3-slim`	3285.3 ± 65.8	3153.1	3367.1
	10000	C	43.4 ± 1.7	41.4	48.9
		`python3-slim`	35945.7 ± 141.3	35722.1	36244.5
	100000	C	54.5 ± 2.0	52.0	59.6
		`python3-slim`	35883.9 ± 101.4	35760.7	36083.8
	1000000	C	160.3 ± 2.4	158.2	164.6
		`python3-slim`	35899.8 ± 211.7	35685.7	36464.5
	10000000	C	1209.8 ± 1.7	1208.4	1214.4
		`python3-slim`	36008.8 ± 276.3	35698.5	36695.0

Table 1: Benchmark Results

B Screenshots

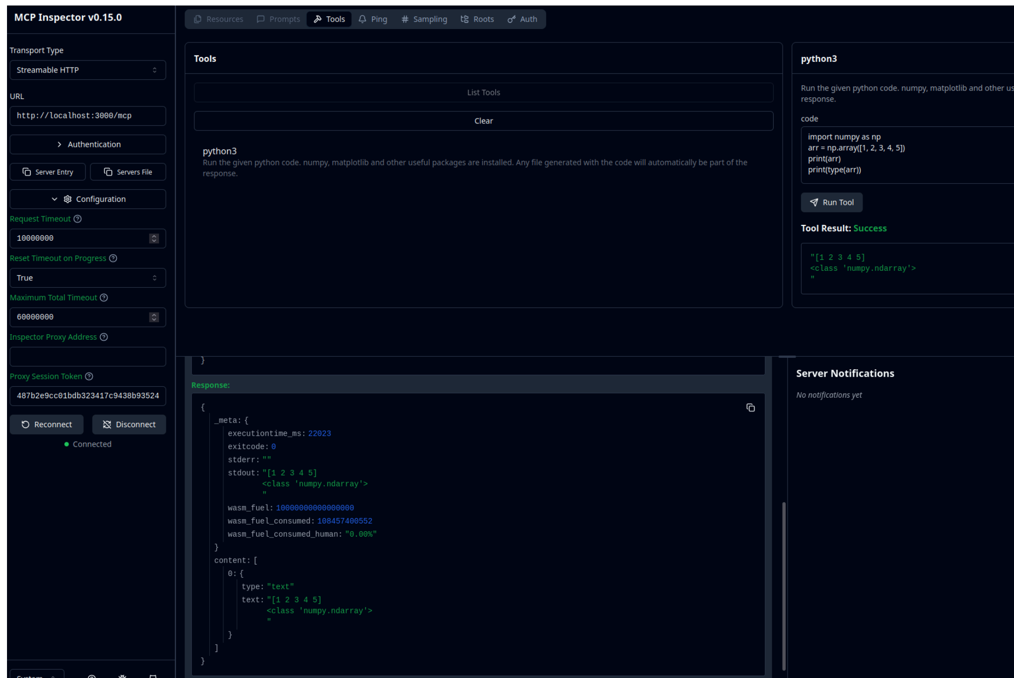

Figure 4: The python3-full tool being used to run four lines of python code to create and print a simple numpy array. The execution time was ca. 22 seconds.

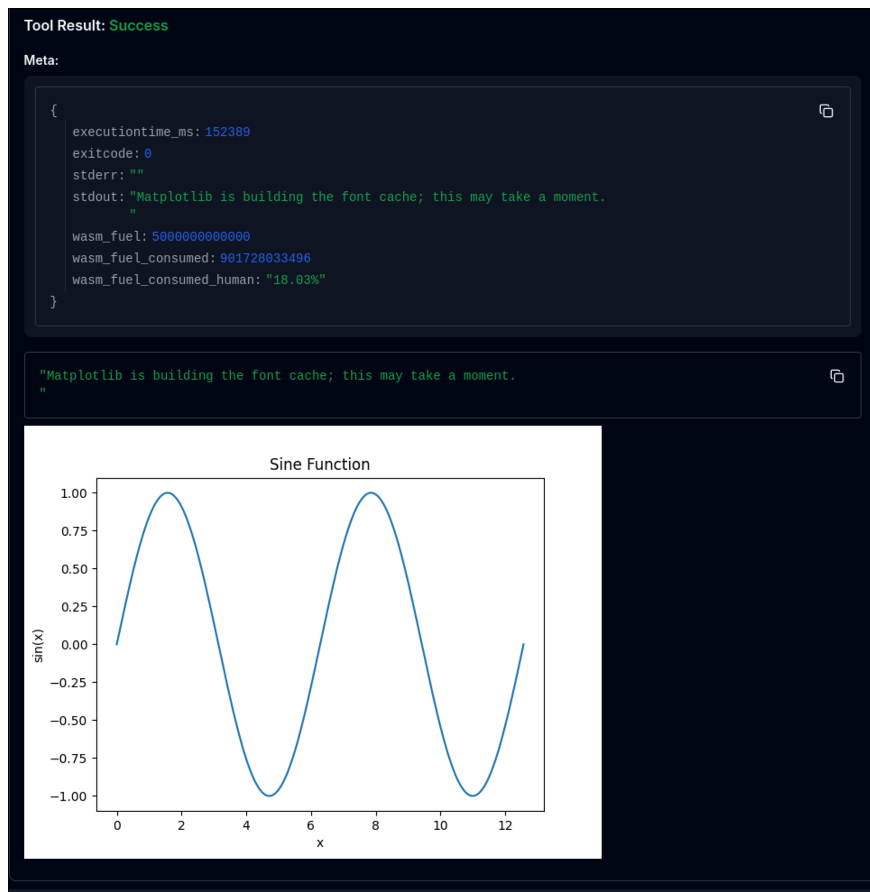

Figure 5: The python3-full tool being used to plot the sine function from 0 to 4π using matplotlib and numpy. The execution time was ca. 2 ½ minutes.