MacTell: an LLM-Based Code Interpreter

Anastasiia Senyk
Applied Sciences Faculty
Ukrainian Catholic University
Lviv, Ukraine
anastasiia.senyk@ucu.edu.ua

Yaroslav Korch
Applied Sciences Faculty
Ukrainian Catholic University
Lviv, Ukraine
yaroslav.korch@ucu.edu.ua

Matvii Prytula
Applied Sciences Faculty
Ukrainian Catholic University
Lviv, Ukraine
matvii.prytula@ucu.edu.ua

Liliana Hotsko
Applied Sciences Faculty
Ukrainian Catholic University
Lviv, Ukraine
liliana.hotsko@ucu.edu.ua

***Abstract*—Mactell is a Swift-powered project designed to integrate multiple functionalities into a macOS application. By employing natural language processing, Mactell interprets user commands into executable actions, significantly improving user interaction within OS and productivity. Key functionalities include command-line interface (CLI) management, web browsing, code execution, media playback, and other OS-accessible tasks, all presented through an intuitive, unified interface.**

I. Introduction

Interacting with various graphical user interfaces (GUIs) and command-line interfaces (CLI) can pose challenges for many users. Issues often arise from the time-consuming process of learning and remembering specific commands, as well as the inherent complexity of terminal syntax. In response to these challenges, our project proposes the integration of a Large Language Model within a user-friendly Swift application designed for macOS.

This application aims to streamline user interaction by offering a unified interface that simplifies the execution of commands. Users can describe their desired tasks using natural language descriptions, which the application interprets into actionable commands. Additionally, users can monitor command execution status and effectively manage workflows by verifying, saving, deleting, and rerunning generated commands.

II. Related Work

A. Open-Interpreter

OpenInterpreter [3] laid the groundwork as a terminal application utilizing a Large Language Model (LLM) to interpret user inputs and execute corresponding commands.

OpenInterpreter offers users the flexibility to select specific LLM models, whether free or paid, and execute generated code based on user prompts. Despite these capabilities, our investigation revealed several usability challenges inherent in terminal-based programs. Consequently, we aimed to develop a dedicated Swift application to enhance user experience by providing a more intuitive interface.

Moreover, concerns about the lack of robust security measures in existing versions, aside from a beta version running code in a container environment, underscored the necessity for enhanced safety protocols in our application. Furthermore, our goal extended beyond basic code execution by exploring additional functionalities enabled by LLM integration (see nest sections for detailed discussion).

B. YAI

YAI [4], an AI-powered terminal assistant, shares similarities with the MacTell in terms of security features, such as user confirmation prompts (”Yes” or ”No”). However, our experiments revealed that constant user confirmation for routine actions (e.g., ”Open Safari”) may hinder user convenience. As a result, we developed a new approach using contextual indicators (green and red functions) to enhance usability, as elaborated in the next sections.

C. IBM – CLAI

IBM’s CLAI (Command Line Instrumentation as a New Environment for AI Agents) [1] presents an interesting approach despite receiving mixed feedback from users and utilizing its own Large Language Model (LLM).

Both CLI and MacTell projects share a common goal: developing custom features aimed at enhancing the usability of LLMs. These features are designed to simplify interactions with LLMs, making them more accessible for everyday use. IBM's approach goes further by creating specialized projects that enable LLMs to leverage additional contextual knowledge, expanding their capabilities.

However, this approach introduces challenges such as longer setup times and the potential complexity of extensive functionalities, which could overwhelm LLMs. In contrast, our project emphasizes the development of streamlined functions that prioritize simplicity and efficiency in interacting with LLMs.

III. Project Development Steps And Ease of Use

A. Initial Concept and Execution

The initial step involved developing a basic system where users could input commands in natural language. The LLM, integrated into the backend, processed this input to understand the user's intent and generate corresponding terminal commands. These commands were then executed, delivering the desired outcome without requiring the user to directly interact with the terminal.

B. Enhanced with Logging and User Profiling

Subsequently, we incorporated a logging mechanism to record each user input and the system's response. This feature creates a comprehensive interaction history, facilitating tracking. Additionally, it personalizes the user experience by adapting to the user's interaction patterns based on the saved messages.

C. Development of a User-Friendly Interface

Recognizing the importance of accessibility and ease of use, the final step was to develop an intuitive interface for the application. This interface, developed in Swift specifically for macOS, offers users a comfortable and visually appealing environment to input their commands in natural language. The application not only displays the history of saved interactions but also supports parallel execution of tasks, an enhancement over the traditional command-line interface (CLI).

IV. Implementation

A. Server

The asynchronous server is the core component of the entire project and connects the Swift application to the backend and database. Its functionality is divided into the following sections:

- Message Listener (main coroutine).
- LLM Client.
- Database Interaction.
- Message Sender.

For a detailed exploration of the server's functionality, refer to the self-developed communication API between the Swift frontend and the server (see Table I on page 4).

The main pipeline of the server includes the following stages:

- **Receiving**: Receives a message from the Swift frontend with the user's request.
- **Confirmation**: Sends a confirmation to the Swift frontend that the request has been received.
- **Input Redirection**: Redirects the user input to the LLM (in a separate coroutine).
- **Processing**: Processes the response once received from the LLM.
- **Response Notification**: Sends a message to the frontend indicating that the response is ready.
- **Execution Without Confirmation**: If the LLM response requires no user confirmation, it is sent directly to the user (or executed first, with the result then sent to the user).
- **User Confirmation Request**: If user confirmation is needed, a message is sent to the Swift application to request confirmation from the user.
- **Execution With Confirmation**: Upon receiving user confirmation, the pipeline continues as described above.

Throughout these actions, the database continuously refreshes its contents based on the current state of the user's request. This operation occurs in a separate coroutine, ensuring non-blocking, thread-safe server communication.

B. Function Calling

For OpenAI models starting from *GPT-3.5-turbo*, the Function Calling technique is used. When the LLM identifies an appropriate function to respond to a user's request, it returns the result in JSON format, mentioning this function and its input.

To enhance user safety, functionalities are categorized into two groups: green for actions considered safe that do not require user confirmation and red for actions that require user consent before execution. Green functions include tasks like searching Google or composing an email (but not sending it). Red functions explicitly inform the user of the intended action, such as sending a message or executing LLM-generated code.

The list of integrated functionalities includes:

Green Functions:

- **Open macOS Application**: Opens the specified macOS application (e.g., FaceTime, Notes, Messages, Safari).
- **Tell Date and Time**: Uses VoiceOver to announce the current date and time.
- **Compose Email**: Composes an email to the recipient and opens the Mail window. The user must press the ”Send” button.
- **Play Music**: Plays the track specified by the user from the user's library.

- **Compose Note**: Creates a note as specified by the user and saves it in the Notes application.
- **Perform Google Search**: Searches Google based on the user's request and opens Safari with the results.
- **Tell Number of Unread Messages**: Uses VoiceOver to announce the number of unread messages.

Red Functions:

- **Make Call by Contact Name**: Calls the specified contact via FaceTime.
- **Make Call by Phone Number**: Calls the specified phone number via FaceTime.
- **Run Generated Code**: Executes scripts generated by the LLM (supports AppleScript, Shell, and Python code).
- **Schedule Command**: Schedules the specified command with a specified frequency.
- **Write Message Using Contact Name**: Writes a message through the Messages app to the specified contact.
- **Write Message Using Phone Number**: Writes a message through the Messages app to the specified phone number.

C. Asynchronous Database Interaction

The asynchronous nature of the server requires appropriate handling of database interactions for different coroutines to avoid data races, where multiple coroutines access the same database row concurrently. We have used global `lock` and `unlock` methods provided by the `asyncio` library to resolve this issue.

In addition to obvious database columns such as user input, a `Status Code` is also stored. This simplifies the data retrieval process and maintains the state of the entire pipeline.

When the application is closed, any ongoing processes initiated by the user can still be completed, and the resulting changes are updated in the database. Upon reopening the application, these updates are fetched, providing the latest pipeline state.

The database also supports direct message fetching, which eliminates the need to resend identical requests to the LLM APIs, making the process faster and cost-free.

D. Application

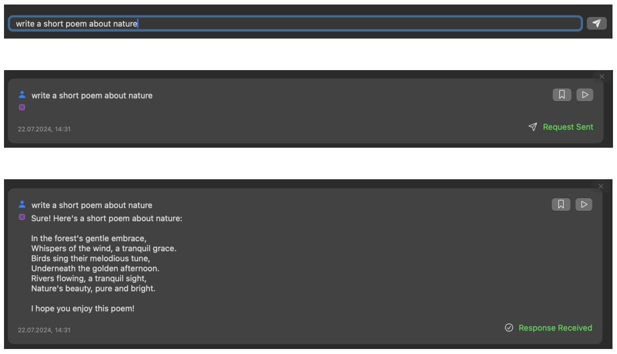
The application has two main windows: `New` and `Saved`.

The `New` window starts empty and is dedicated to the current running process. If the application closes, all ongoing conversations in this window are lost, and corresponding entries are deleted from the database.

Each message pair (bubble) in this window includes two buttons allowing users to save it to the `Saved` window. If the message is in both windows, only one bubble instance is in the memory.

The second button is either `Rerun` or `Confirm`. The `Rerun` button allows the user to rerun the previously generated (by the LLM) code or completely regenerate the LLM's response, which depends on what the LLM has returned. For example, if the user asks to ”write a poem,” the LLM response will be regenerated completely. But if he asks: ”Write the script that adds two numbers. Add 2 and 2.”, only the script's output and execution status (successful or with errors.) will be displayed. Examples of the interface are presented in Fig. IV-D, IV-D, IV-D.

The application supports the simultaneous execution



write a short poem about nature

write a short poem about nature

22.07.2024, 14:31

write a short poem about nature

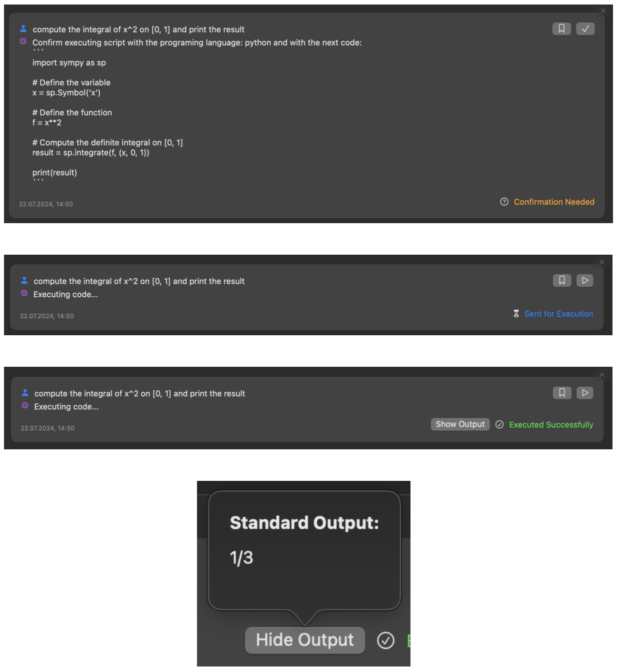
Sure! Here's a short poem about nature:

In the forest's gentle embrace,
Whispers of the wind, a tranquil grace.
Birds sing their melodious tune,
Underneath the golden afternoon.
Rivers flowing, a tranquil sight,
Nature's beauty, pure and bright.

I hope you enjoy this poem!

22.07.2024, 14:31

Fig. 1. Example of the interaction with the application. Start with a prompt, explore the generation status, and find the result in a few seconds.



compute the integral of x^2 on [0, 1] and print the result

Confirm executing script with the programing language: python and with the next code:

```
import sympy as sp

# Define the variable
x = sp.Symbol('x')

# Define the function
f = x**2

# Compute the definite integral on [0, 1]
result = sp.integrate(f, (x, 0, 1))

print(result)
```

22.07.2024, 14:50

compute the integral of x^2 on [0, 1] and print the result

Executing code...

22.07.2024, 14:50

compute the integral of x^2 on [0, 1] and print the result

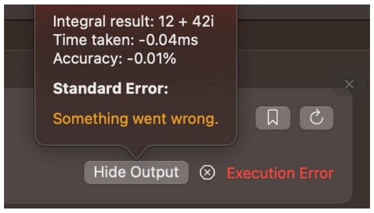
Executing code...

22.07.2024, 14:50

Standard Output:

```
1/3
```

Fig. 2. Code generation examples.


```
Integral result: 12 + 42i
Time taken: -0.04ms
Accuracy: -0.01%

Standard Error:

Something went wrong.
```

Fig. 3. Case of the error.

of pipelines due to the server's asynchronous nature. Almost every `Status Code` received from the server (or sent from the application) updates the UI to reflect the pipeline's state. This enables users to monitor the status of their tasks effectively.

For detailed project documentation, refer to [2].

V. Testing

- **Crash Test.** We have tried to spam the server with multiple messages from the app to test for potential concurrency problems.
- **Speed.** On average, the ChatGPT 3.5-turbo model processed the user's response in 2 seconds. However, the database fetching (when `Rerun` pressed) simplifies this step and allows us to immediately create a new process for the execution (if needed).
- **Suggestions.** LLM Suggestions – a way for an LLM to notify a user about possible improvements in the methods a user asks. After long tests, we found that the best models currently cannot provide suggestions for the user, even if his request is strange. LLM hallucinates more than brings useful information.

VI. Possible Improvements

The newest versions of LLMs from OpenAI have introduced the capability to invoke combinations of functions using a function-calling methodology. This enhancement offers a promising way for our project, requiring straightforward adjustments in how we parse and utilize responses from the LLMs. By leveraging this feature, our application could seamlessly integrate more complex functionalities and provide richer user interactions.

Current LLM APIs do not support for retaining context, meaning they respond based solely on immediate user input without remembering previous interactions. Overcoming this limitation could enhance the natural flow and continuity of conversations in our application, making interactions feel more coherent and human-like.

Another area for improvement is expanding our application's support beyond macOS applications integrated with AppleScript. Our project architecture is flexible enough to accommodate new functions tailored to specific tasks and requirements. For example, adding functions like `write_message_via_signal` could extend our application's capabilities to platforms lacking native AppleScript support.

VII. Conclusion

In conclusion, MacTell transforms MacOS interaction by integrating a Language Model into a Swift-based application. This allows users to effortlessly execute commands using natural language for functionalities such as CLI operations, web browsing, media control, and others within a unified platform. The backend's asynchronous architecture ensures responsive command execution, optimizing the overall efficiency. Also, constant advancements in LLM capabilities will expand the range of potential functionalities and services provided by our application.

Appendix A

TABLE I
Comprehensive list of status codes used in MacTell communication protocol.

Code	Name	Meaning	Sender
-1	noActionTaken	Default. Placeholder	Server
0	sentForExecution	User request sent to execution	Server
1	askConfirmation	Need confirmation for execution from user	Server
2	requestSentToAPI	Request sent to LLM	Server
3	submitUserResponse	The app tells to submit user input to LLM	App
4	askRerun	User requests the rerun	App
5	rawText	The LLM response is pure text	Server
7	serverCrash	Server crashed	Server
8	confirmExecution	User confirmed execution	App
10	executedSuccessfully	Command executed successfully	Server
11	executionError	Command executed with errors	Server
15	saveToBookmarks	Save user request to Bookmarks	App
16	removeFromBookmarks	Remove user request from Bookmarks	App
18	deleteUserMessage	Delete user request entirely from the Application	App
19	askAllSaved	Retrieve all saved requests from DB	App
20	sendAllSaved	Send all saved requests from DB	Server

References

[1] M. Agarwal, J. J. Barroso, T. Chakraborti, E. M. Dow, K. Fadnis, B. Godoy, M. Pallan, and K. Talamadupula. Project clai: Instrumenting the command line as a new environment for ai agents. *ArXiv*, 2020. https://arxiv.org/abs/2002.00762.
[2] Y. Korch, M. Prytula, A. Senyk, and L. Hotsko. Interpreter documentation. https://github.com/NaniiiGock/Interpreter, 2023.
[3] Valerie. Open interpreter — a fantastic tool that will allow ai to run code on your computer. Medium, *Dare To Be Better*, https://medium.com/dare-to-be-better/def7eef2d211, September 2023.
[4] Jonathan Vuillemin. Yo: Ai powered terminal assistant. Medium, https://medium.com/@jovllmn/yo-ai-powered-terminal-assistant-958408e8f1c7, April 2023.