OS Project Report

Introduction and motivation

One of the problems in current Linux systems is that users have to memorize commands and run those commands, but sometimes they forget what to write, and sometimes, in order to do some popular tasks like investigating usage and getting processes that use resources the most, they need to run a sequence of tasks, which is hard for those who are not experts. Also sometimes users run commands without knowing the severity of those commands, so our job here is to let them know the severity of commands they are running, and prevent unauthorized users from running privileged commands, so that is why we decided to build OLLMCPC which uses LLMs like Gemini, so the user will not have to write commands, but just chat with the LLM (something like SIRI) with a security layer that precedes commands execution that makes sure that the user is authorized, and that the user has no problem in executing commands that appear to be risky like deleting files.

Overall Project Architecture

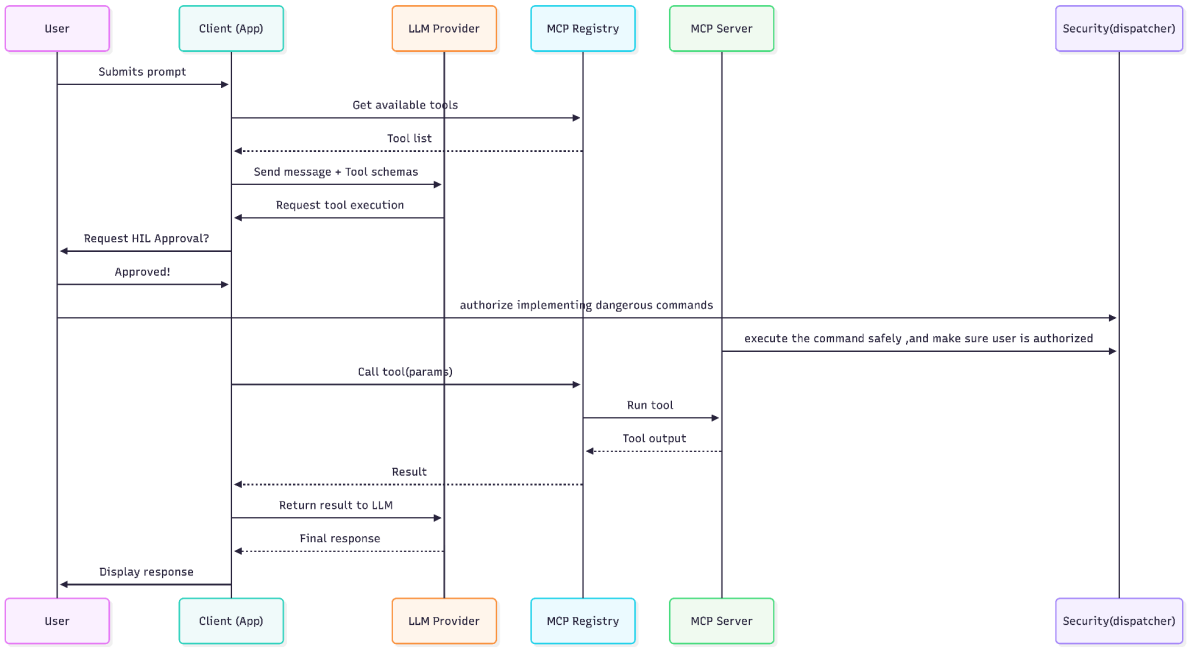

Fig1. Overall Project Architecture

For More Details check out the Documentation:
https://mohammed-alaa40123.github.io/ollmcpc_v2/

The system consists of four main layers:

1. **Application Layer**: Handles the CLI, Terminal UI (TUI), and configuration management.
2. **LLM Layer**: A set of provider strategies that translate general tool/chat requests into provider-specific API calls (Ollama, Gemini).
3. **MCP Layer**: Implements the Model Context Protocol. It manages server lifecycles, tool discovery, and execution.
4. **Security Layer**: This layer makes sure that the user is authorized and makes sure that risky commands are executed with admin permission.

LLM Models Used

This project integrates two Large Language Model (LLM) solutions, Ollama and Google's Gemini API, to provide AI-driven language capabilities. The system is designed to allow the end user to explicitly choose which model to use at runtime. As a result, the project does not enforce a predefined preference or automatic selection logic between models. Instead, both models are exposed through a unified interface, and the user's choice determines which backend processes the request.

Ollama is integrated as a locally hosted LLM runtime that enables running open-source language models directly on the user's machine or a self-managed server. It provides a lightweight service that manages model execution and exposes a simple HTTP-based API. Through this API, the application sends prompts and receives generated responses in a standardized format. Ollama abstracts much of the complexity involved in running local models, such as model loading and inference management, making local LLM usage straightforward to integrate.

The primary strengths of Ollama lie in its local execution model. Because inference runs entirely on the host system, no user data is transmitted to external services. This provides strong privacy guarantees and allows the system to operate without an internet connection. Ollama also avoids usage-based costs, since models are executed locally rather than through a paid API. Additionally, it supports a variety of open-source models, giving users flexibility in choosing the underlying model that best suits their needs.

However, Ollama also has limitations that stem from its reliance on local hardware. Model performance and response time depend heavily on available system resources such as CPU, GPU, and memory. On lower-end machines, inference can be slower compared to cloud-based solutions. Furthermore, locally hosted models may have smaller context windows and may not always match the reasoning quality or consistency of large-scale cloud models.

Gemini is integrated through Google's cloud-based Gemini API, which provides access to advanced large language models hosted on Google's infrastructure. The application communicates with Gemini by sending authenticated HTTP requests containing user prompts, and the API returns generated responses. All model execution and scaling are handled remotely by Google, allowing the application to focus solely on request handling and response processing.

The main strengths of Gemini are its high-quality language generation, strong reasoning capabilities, and ability to handle complex and lengthy prompts. Because it runs on managed cloud infrastructure, performance is consistent and does not depend on the user's local hardware. Gemini also benefits from continuous improvements and optimizations provided by Google, ensuring access to up-to-date model capabilities.

At the same time, Gemini has inherent constraints associated with cloud-based APIs. It requires an active internet connection, and usage is subject to API quotas and cost considerations. Additionally, user inputs are sent to external servers for processing, which may be a concern in scenarios involving sensitive data. These characteristics are clearly documented so that users can make informed decisions when selecting the model.

Now, given that both models have their strengths and weaknesses, we give the user the choice to choose between any of them or work in manual mode. So, overall, the system flow will include two paths to execute commands, either through stdio or LLMs, as shown in Fig 2.

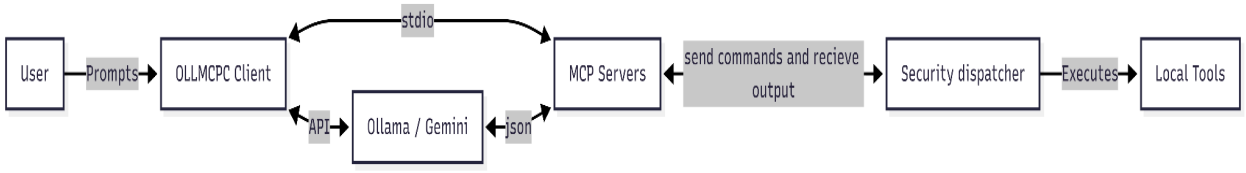

Fig2. Overall System Flow

Security Layer

This layer gets the raw commands from the MCP server, checks on commands passed, makes sure that commands are authorized for the user, and runs the commands in child processes to isolate their execution from the main program. The logic for this is later in the dispatcher C program, and in the coming paragraphs, there is a detailed explanation of the logic in this file.

The Dispatcher C program implements a secure command dispatcher that behaves like a small user-space “kernel” or protection boundary between a user and the underlying Linux system. Its primary goal is to classify commands, enforce privilege policies, warn against dangerous operations, log all actions, and then execute commands safely. The program supports both internal commands implemented in C and external Linux binaries, while applying different levels of security control depending on the command type and its risk level.

At the core of the design is a privilege model defined using the PrivilegeLevel enumeration. Commands are categorized as normal user commands, administrative commands, or dangerous commands. This classification determines how the dispatcher treats each command before execution. The CommandMetadata structure stores all security-relevant information for a command, including its name, privilege level, optional warning message, whether it is internal or external, and a function pointer for internal commands. This allows the dispatcher to uniformly process different kinds of commands using a single policy framework.

A key security feature of the program is user authorization. The is_authorized_user function verifies whether the current user is root or belongs to a designated administrative group (defined as "sudo"). This is done using standard Linux user and group lookup APIs. Administrative and dangerous commands are blocked immediately if the user lacks sufficient privileges, preventing unauthorized access to sensitive operations.

The figure below shows a generic flowchart explaining the flow of dispatcher and cases for a command to be executed ,and how commands are executed

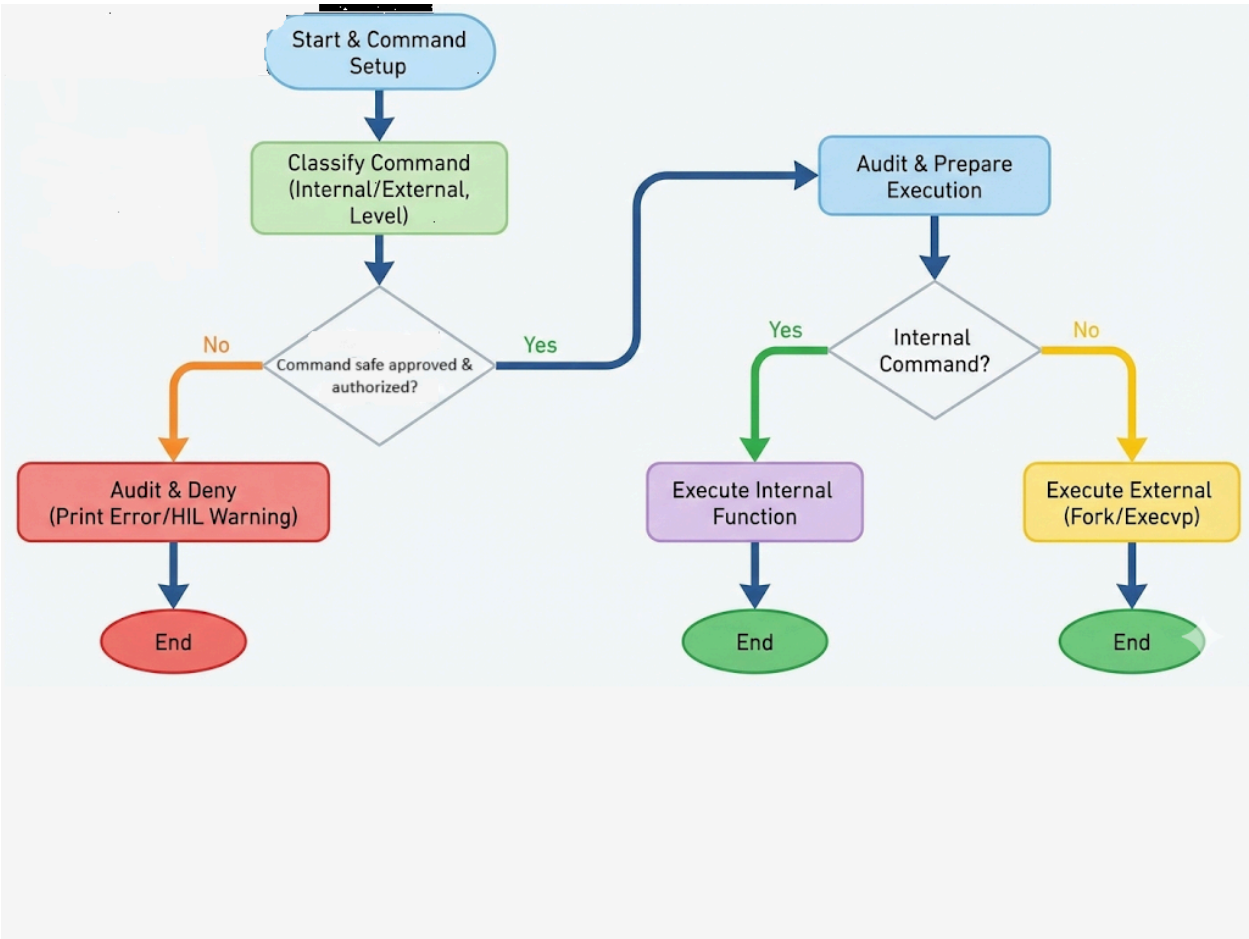

Fig 3 . Dispatcher Security Diagram

COMMANDS AND SCRIPTS

We had multiple different scripts we ran throughout the project that we send to our vector tools metadata in the server app C++ layer. These scripts were designed to perform many different operating system tasks, mainly related to process management and control, signal manipulation, threads, memory and resource management, and general system status, we relied on these scripts as the tools the LLM uses to quickly and reliably answer the user.

OS Assistant & Quick Health Checks

Tool	What it does
osassist_battery_info.sh	Battery status plus a quick memory snapshot. A pulse check before you start changing anything.
osassist_memory_info.sh	Memory and swap summary with battery context so you can tell ‘under pressure’ vs ‘just busy’.
osuptime_plus.sh	Uptime + load averages, plus a quick look at top CPU and memory processes for instant triage.
osmem_usage.sh	Memory and swap status line for quick health checks (nice before/after comparisons in the report).

User Identity & Shell Experience

Tool	What it does
oswhoami.sh	Who you are, where you are, and the terminal session you are in, helpful for permissions and debugging.
osecho_plus.sh	Prints a message with optional level and timestamp, so your outputs look consistent and professional.
oshelp.sh	Mini-OS help that lists commands and quick examples, your 'home page' for demos.
osrepl.sh	Mini-OS interactive shell (manual run). A persistent prompt that makes the project feel like an OS environment.

Filesystem Search & Monitoring

Tool	What it does
osfind.sh	Find by name pattern with basic type filtering; handy for locating configs, logs, or assignments quickly.
osgrep.sh	Grep by pattern and path with safe defaults. Great for 'find the error line in a log' moments.
osfile_watch.sh	Watches a file or directory for size and modified-time changes. Demonstrates loops and live monitoring.
osdir_size_top.sh	Largest items in a directory so cleanup targets stand out immediately.
osdiskfree.sh	Disk usage summary for a path with a low-space warning to catch problems early.

Process Inspection & Control

Tool	What it does
osps.sh	CPU-sorted process list for fast triage, answers ‘what is using my machine right now?’
process_info.sh	Detailed ps line for a PID (CPU/MEM/state/runtime) when you need specifics.
process_state.sh	Compact process state view for a PID (running/sleeping/zombie, etc.).
osproc_find.sh	Find processes by name pattern with a compact table. Fast way to locate the right PID.
osproc_openfiles.sh	Open files for a PID via lsof or /proc fallback. Shows what resources a process is using.
osstop.sh	Pause a process safely (SIGSTOP), like putting a task ‘on hold’.
oscont.sh	Resume a stopped process (SIGCONT) brings the task back to life.
oskillsafe.sh	Gentle termination: confirmation + SIGTERM first, with a fallback to SIGKILL if needed.

Process Trees, Forking, Zombies & Orphans (Demos)

Tool	What it does
osproctree.sh	Process tree starting from a PID, visualizes parent/child relationships clearly.
osproc_children_list.sh	Child processes of a PID in a simple tree, focused on descendants only.
osspawnchildren.sh	Spawns N child processes to demonstrate branching from a parent and tracking child PIDs.
ostreedemo.sh	Fork tree demo with clear parent-child relationships in action.
oszombie.sh	Zombie demo that surfaces Z-state processes and explains why they happen.
osorphan.sh	Orphan demo that highlights PPID changes after the parent exits.

Signals & IPC

Tool	What it does
ossig_pingpong.sh	Signal ping-pong demo for IPC basics: parent/child exchange signals for a few rounds.
run_shell_command.sh	Executes a shell command, useful but treat with care; a good place to discuss safety rules in the report.

Memory Maps & Shared Memory

Tool	What it does
osmem_heapstack_range.sh	Heap/stack ranges from /proc/PID/maps, connects memory concepts to real processes.
osshm_list.sh	Shared memory segments list (ipcs -m). Useful to prove shared memory exists and to sanity-check demos.

Threads

Tool	What it does
osthread_demo.sh	Thread roles demo for quick concurrency visuals.
osthread_sync_demo.sh	Thread sync demo that shows why coordination matters.