Palestine Polytechnic University

College of Information Technology and Computer Engineering

Department of Computer Science

Autopilot: Automating System-Admin Tasks With Generative AI

Project Team:

Mohammed Hijazi 211034

Mohammed Nahawi 211018

Momen Hashlamoon 211170

Supervisor:

Dr. Mousa Alrefayah

This project is submitted to fulfill the requirements for a Bachelor's degree in Computer Science. It reflects the knowledge and expertise gained over the course of the program.

Jan, 2025

Acknowledgment

In the name of Allah, the Most Beneficent and Merciful, who gave us strength, knowledge, and helped us get through this project, we dedicate this project to the souls of the martyrs and our brothers and sisters in Gaza.

In addition, we would like to express our immense gratitude to Dr. Mousa Alrefayah for aiding us in this project. Moreover, we owe an immense debt of gratitude to our families, whose unwavering encouragement and continuous support have been the cornerstone of our journey. Their generosity, both in spirit and action, has shaped the very fabric of who we are. All our family members, your belief in us has been a guiding light, and for that, we are profoundly thankful.

Abstract

The role of system administrator requires performing a wide range of tasks related to networking, storage, security, file-system, and more. One challenge faced by system administrators is when working on complex tasks that require executing complex commands with long arguments, another obstacle is when working with continuous redundant tasks that need specialized automation. To relieve this burden, our project aims to develop an automation system powered by AI Agents that specialize in areas such as networking, user and group management, file system, etc. These agents are responsible for taking care of executing the tasks of system admin.

الملخص

يتطلب دور مسؤول النظام أداء مجموعة واسعة من المهام المتعلقة بالشبكات، والتخزين، والأمان، وتنظيم الملفات، وغير ذلك. ومن التحديات التي يواجهها مسؤولو النظام هي عند العمل على مهام معقدة تتطلب تنفيذ أوامر معقدة و طويلة .

كما يواجهون عقبة أخرى عند التعامل مع المهام المتكررة المستمرة التي تحتاج إلى أتمتة متخصصة. لتخفيف هذا العبء يهدف مشروعنا إلى تطوير نظام أتمتة مدعوما ب (AI agents) متخصصة في مجالات مثل الشبكات، وإدارة المستخدمين والمجموعات، وتنظيم الملفات وغيرها. سوف يعمل النظام الذي سوف نبنيه على تنفيذ مهام مسؤول النظام بشكل آلي وترجمة كلام المستخدم الطبيعي الى أوامر يفهمها الحاسوب لتنفيذها.

Contents

1 Introduction **12**

1.1 Overview 12

1.2 Problem Statement 12

1.3 Project Aim 13

1.4 Objectives 13

1.5 System Description 14

1.6 Project Limitations/Constraints 15

1.7 Time Table 16

2 Theoretical Background **17**

2.1 Overview 17

2.2 The Linux Operating System 17

2.2.1 Why Linux 17

2.2.2 Linux Architecture 17

2.3 Artificial Intelligence 18

2.3.1 Natural Language Processing 18

2.3.2 Large Language Models 19

2.4 AI Agents 19

2.4.1 Prompt Templates 19

2.4.2 Tool Calling 20

2.4.3 Chains 20

2.4.4 Agentic Workflows 21

2.5 Literature Review 22

3 System Design and Analysis **25**

3.1 Preface 25

3.2 Requirements 25

3.2.1 Functional Requirements 25

3.2.2 Non-functional Requirements 26
3.3 System Architecture 26
3.3.1 Agents Description 26
3.3.2 Agentic Architecture 28
3.3.3 Workflow of Agents 29
3.3.4 Analysis and Diagrams 30
3.4 Technologies and Frameworks 41

4 Implementation and Evaluation **43**
4.1 Handling Alternatives 43
4.2 Building The Agents 44
4.2.1 AI Models Used 44
4.2.2 Models Specification 46
4.3 Building The Tools 46
4.3.1 Creating Tools 46
4.4 Tasks & Jobs Feature 48
4.4.1 Creating Tasks & Jobs 48
4.4.2 Running Tasks & Jobs 49
4.4.3 Tasks API Endpoints 51
4.5 Dashboard Feature 51
4.5.1 Monitoring Screen 51
4.5.2 Agents Dashboard 52
4.5.3 Dashboard API Endpoints 53
4.6 Chatbot Feature 53
4.6.1 Interfaces & Descriptions 53
4.6.2 Chat-Bot API Endpoints 55
4.7 Auto-Correction Feature 55
4.8 Evaluation 57
4.8.1 Evaluating Modules 57
4.8.2 Evaluation Statistics 59

4.9 Benchmarking 60
4.9.1 Methodology 60
4.9.2 Results 62

5 Conclusion **63**

5.1 Summary 63
5.2 Future Work 63
5.2.1 User Analytics Dashboard 63
5.2.2 Logs System 63
5.2.3 Task Scheduling 63

List of Figures

1 Context Diagram 14
2 Linux Architecture 18
3 NLP illustration 18
4 Prompt Template Example 20
5 Chains illustration 21
6 Plan Data Structure illustration 28
7 Master Slave Architecture 29
8 ReACT Architecture 29
9 Use Case Diagram 30
10 Prompting Sequence Diagram 40
11 Agent Activation Sequence Diagram 40
12 Task Execution Sequence Diagram 41
13 Llama3.2 Agent Integration 44
14 LinuxGeneral Agent Integration 45
15 Llama3.1 Agent Integration 45
16 DeepSeek-R1 Integration 46
17 Tool Code Sample 47
18 Binding Tools With LLM 47
19 Tasks 48
20 Jobs 48
21 Generating Task From Prompt 49
22 Generated Jobs 49
23 Generated Task 49
24 Executing Jobs 50
25 Monitoring Screen 52
26 Agents Dashboard 52
27 Agent Description 52

28 Chat-Bot Summarization . . . 54
29 Taking Approval From User . . . 54
30 Max Iterations . . . 56
31 benchmark the system . . . 60
32 benchmark_system() function . . . 61
33 write_benchmark() function . . . 61

List of Tables

1 Project Time Table 16
2 Summary of Literature 22
3 Use Case: Send Prompt 31
4 Use Case: Create Task 32
5 Use Case: Generate Task 33
6 Use Case: Delete Task 34
7 Use Case: Update Task 34
8 Use Case: Disable Feedback 35
9 Use Case: Run Feedback 35
10 Use Case: Run Agent 36
11 Use Case: Disable Agent 36
12 Use Case: Generate Response 37
13 Use Case: Execute Commands 38
14 Use Case: Send Command/Job Status 38
15 Use Case: Send Resource Usage 39
16 Use Case: Send Confirmation Message 39
17 Handling Alternatives 43
18 Models Specification 46
19 Tasks API Endpoints 51
20 Dashboard API Endpoints 53
21 Chat-Bot API Endpoints 55
22 Network Tasks Evaluation Table 57
23 Users & Groups Tasks Evaluation Table 57
24 Software Packages Evaluation Table 58
25 Software Development Evaluation Table 58
26 Filesystem Evaluation Table 58
27 Processes & Services Evaluation Table 58

28 Success & Failure Rates Table 59
29 System performance metrics for different tasks 62

1 Introduction

1.1 Overview

This chapter serves as a brief introduction to the project. Through this chapter, we intend to explain the overall description of the system along with what we aim to achieve and the problem we are solving.

1.2 Problem Statement

We can generally divide the challenges faced by system administrators into three main challenges:

First, since there is a need to manage and maintain different server modules, this creates the need to know and remember a huge number of commands and their arguments, not to mention remembering how to execute complex commands. The second challenge is performing redundant tasks that might be simple sometimes but are time-consuming. These repetitive tasks, like updating system packages, creating multiple users, groups, and permissions, might seem simple and straightforward, but the frequency of these tasks can divert the sys-admin's attention away from more critical tasks. Third, continuously keeping up with new technologies and adopting them into the system requires a lot of dedicated time for learning and integration.

Through this project, we aim to eliminate these challenges by building AI agents, each handling an OS module (e.g., Networking, Managing Users and Groups, etc.). After that, we distribute the sys-admin tasks among them and let them complete these tasks. By doing so, we can finish tasks by writing in natural language instead of writing complex commands. Additionally, since the tasks are automated using AI, we can save time instead of manually performing repetitive tasks.

1.3 Project Aim

The project aims to introdue a new way of managing operating system and change the way the user interact with operating systems by developing automation system for sys-admin tasks, these tasks will be automated using AI agents. This approach will reduce time-consuming operations and save time by automating repetitive tasks, enhancing productivity and user experience. The user can interact with the system by providing a centralized panel/interface for controlling the OS. The project will abstract the complexity of managing operations systems by using natural language then converting it to automated tasks.

1.4 Objectives

The specific objectives of the project are as follows:

1. Design a central web interface to interact and assign tasks to Agents.
2. Build AI Agents that manage different OS modules.
3. Design a workflow of interaction between these Agents.
4. Develop an Interface to monitor and control Agents actions.
5. Test and Evaluate the effectiveness of Agents through running tasks.

1.5 System Description

The core of the system is the AI agents, which are composed of large language models that have access to the Linux shell to execute commands. Additionally, the system consists of four external parts that interact with the core of the application. The Redis database is one of the main external parts of the system. It is used to read, write, and modify the tasks assigned to each agent. Furthermore, we store the state of each agent and record more information about running and completed tasks. The Linux shell is the most critical part of the system, where agents send commands to be executed and receive the output of these commands. The system also makes use of external APIs that connect the forntend with the backend. Finally, the end user interacts with the system through a web panel to assign tasks or through a Tasks Panel or using a Chat-bot for executing quick tasks.

Figure 1 shows context diagram of the system and the outside interactions :

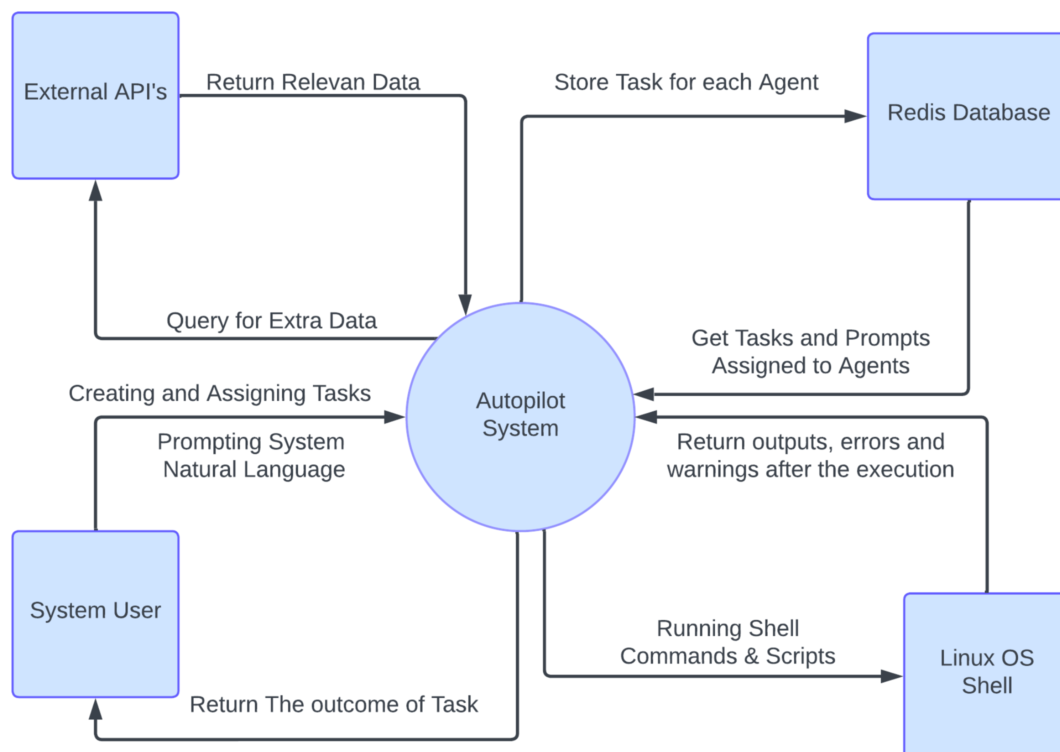


Figure 1: Context Diagram

1.6 Project Limitations/Constraints

Certain limitations and constraints need to be acknowledged when building our project:

- **Hardware Constraints:** The Agents will make decisions using small to medium-sized LLMs (Large Language Models), which may result in less accurate responses and results. This is because we need the agents to run on our local laptop. (will be dicussed in chapter 4, see Table 18: Models Specification)
- **Concurrency Limitations:** The hardware we are using is capable of running at max 2-3 models concurrently before showing performance issues, this will limit the number of tasks running concurrently.
- **Environment Limitations:** The system will be running exclusively on Linux Operating System, since other systems don't provide the flexibility and the control that the Linux systems provide.
- **Multi-Tasking Limitations:** The system is capable for running multiple task all at the same time but since we are running the model locally this might create an performance overhead when it comes to gpu and cpu utilization, a proposed solution for this issue is to take advantage of online cloud hosted models when the local ones are in use.

1.7 Time Table

Table 1: Project Time Table

	The summer semester			The First semester			
Week	1	2-3	4-9	1 - 4	5 - 10	11 - 14	15
Selection of project Idea							
Collecting the Data							
System Design							
System Implementation							
System testing							
Documentation							

The project starts with one week of brain storming to come up with an idea for the project followed data and information collection through weeks two and three then we start system design while also still doing some data collection in the weeks from 4 to 9. The implementation phase starts in the second semester weeks from 1 to 4 we finalize the system design and start to implementing the project, after week 10 we begin testing the system and finalize the implementation phase. The process of documentation carries from the start of the project until finishing everything in it.

2 Theoretical Background

2.1 Overview

This chapter provides the theoretical background needed to understand the different principles and technologies we are going to use for developing the system.

2.2 The Linux Operating System

2.2.1 Why Linux

Linux is an open-source operating system that is famous out for its unparalleled flexibility, customizability, and scalability. The ability to use command line tools and commands to fully and thoroughly manage and control the OS makes Linux the right tool for the job[3] And if text-based commands are used to manage the OS we can take advantage of this by giving the Large Language Models the ability to output text-based commands into the terminal that reflect the user intent.

2.2.2 Linux Architecture

Like a typical Operating System the Linux architecture consist of multiple layers as Figure 2 shows. The kernel is responsible for managing resources like cpu and memory and allowing indirct communication between processes and hardware. The Shell is a command-line interface for communicating with the kernel. Finally GUI layer which includes Graphical Applications.[6]

Understanding the Linux Architecture is vital for building our system. We are mostly interested in the GUI layer and the Shell layer. The Shell layer is used by the AI Agents to run commands on the system and the GUI will display a Web interfaces that shows the results of execution.

Figure 2 displays the Linux OS architecture and its various components interacting with the system.

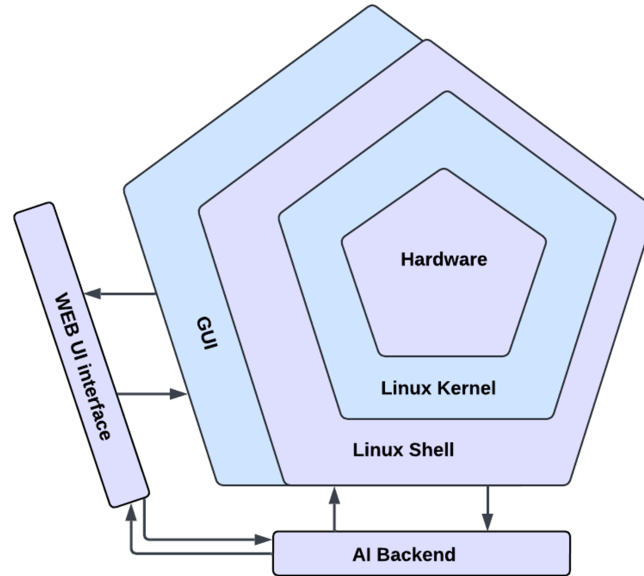


Figure 2: Linux Architecture

2.3 Artificial Intelligence

2.3.1 Natural Language Processing

Natural Language Processing (NLP) is considered a sub field in artificial intelligence which deals with communication between computers and humans language. It focuses on enabling computers to understand and generate natural language as text or speech.[7] In this project we will use Large Language Modelts to convert Natural Language into commands to execute on Linux.

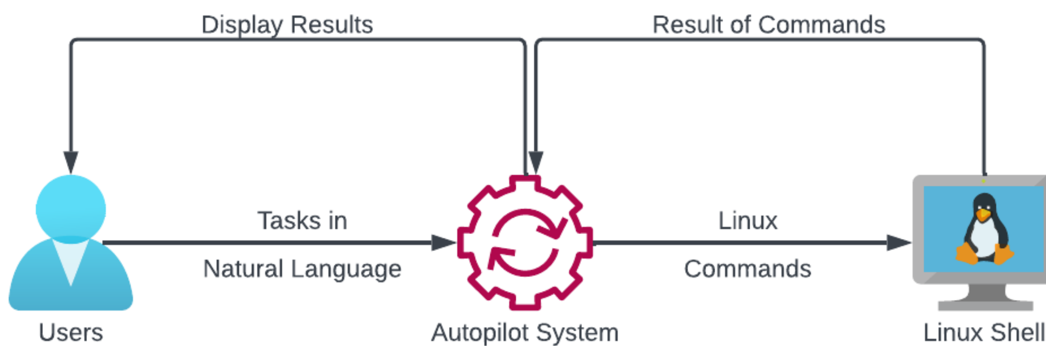


Figure 3: NLP illustration

2.3.2 Large Language Models

Large Language Models (LLMs) are models trained on large amounts of text data to learn patterns and relationships between words, allowing them to generate coherent and human-like text. LLMs can be used for a variety of tasks, such as language translation, question answering, and text summarizing.[2] In this project, LLMs will serve two main purposes: The First we will use open source models trained on Linux datasets which will be used to suggest the right commands for the user tasks. The Second is decision making, after executing a command on the shell these models will examine the output and decide wether it was successful in completing the task or wether there is a need to run other commands.

2.4 AI Agents

AI Agents represent a system where LLMs can access external environments and make real decisions based on info coming from external systems. For example an AI Agent for the stock market would receive information about a certain stock going up from external API's this information will be analyzed by LLM inside the agent and then the LLM will suggest to sell the stock by providing the name of the function of API endpoint that sells that stock. we take the suggestion from LLM and execute it. In Summary Agents can be used to automate tasks, provide user support, or facilitate decision-making.[8]

2.4.1 Prompt Templates

Prompt templates are just a static placeholder for the user input, they include some instruction targeted towards the LLM itself to optimize the response of the AI model to fit the users needs. The next figure is a simple example of utilizing prompt templates to tweak the response we want to get from the LLM. In the example, we optimize the responses coming out of the AI model to be only in Arabic, we do so by creating a prompt template then augmenting the users input inside it then sending it to the model.

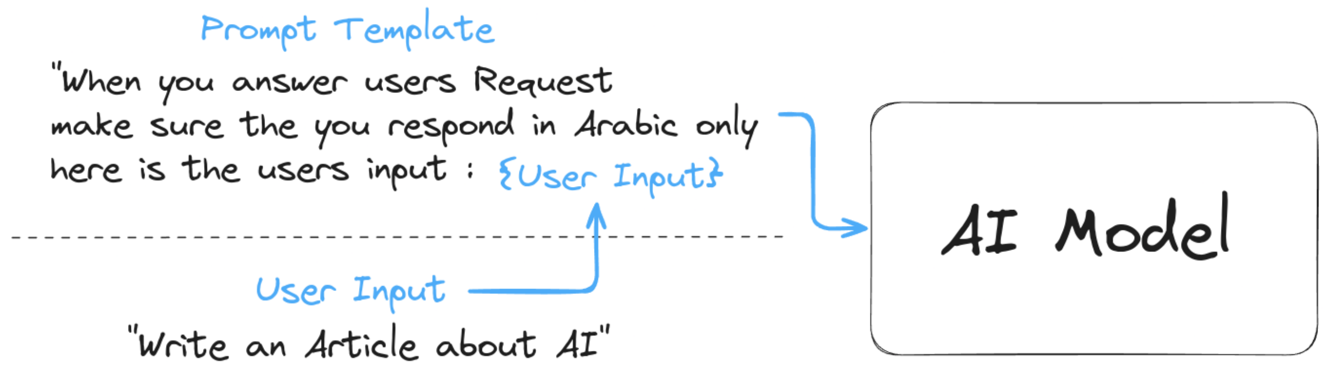


Figure 4: Prompt Template Example

2.4.2 Tool Calling

Tool Calling is a new technology the enables the LLM to communicate with outside environments and make decision to execute functions that effect the outside environment. The way tool calling works is by taking in the user prompt and before giving it to the LLM we add extra context about the existence of Tools (functions or API's) that can be used to satisfy user request. if one of the tools can achieve the users request the LLM will output the name of the function that should be used and the arguments that we should to the function. Finally the function is executed.

2.4.3 Chains

chains are a sequence of components or steps that are dictated by the developer to reach a response suitable for the system they are developing. the chain is linked together to perform one or more complex tasks. Each component in the chain can be a model, a prompt, a parser, or another chain, and they work in tandem to process input and produce the desired output.[4]

Figure 5 illustrates chaining process.

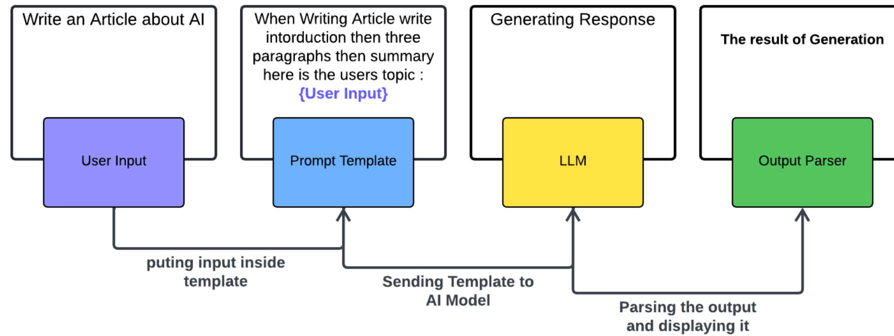


Figure 5: Chains illustration

2.4.4 Agentic Workflows

Workflows refer to the overall process and small steps related to agent execution. Effective workflow design can significantly impact the responses of LLMs. By optimizing workflows, developers can shape an accurate and robust agentic systems, and enhance the overall user experience.

The workflow of agents can be an iterative, interactive process, where the AI agent reflect on the results of executing a task and chooses what to do next. This contrasts with the traditional "non-agentic" approach, where AI models are simply prompted to generate outputs in a single pass. [5]

There are many Design Patterns used to achieve a great flow some these are :

- **Reflection :** The core idea is that the AI agent can examine its own work and identify ways to improve it, rather than simply generating output in a single pass.
- **Tool Use :** Integrating functions and external tools to manipulate, and gather information from outside environments.
- **Agents Collaboration :** Agentic workflows can involve multiple AI agents working together, each playing a different role, to arrive at better solutions than a single agent could.
- **Planning :** which includes letting the agent generate a plan before generating a final response.

2.5 Literature Review

Table 2: Summary of Literature

Title	**Authors**	**Year**	**Main Idea**
Managing Linux servers with LLM-based AI agents: An empirical evaluation with GPT4.	Charles Cao, Feiyi Wang, Lisa Lindley, Zejiang Wang	2024	"An empirical study on the application of LLM based AI agents for automating server management tasks in Linux environments. Aims to evaluate the effectiveness, efficiency, and adaptability of LLM-based AI agents in handling a wide range of server management tasks, and to identify the potential benefits and challenges."[1]
CLI Command Generation Using Generative AI	Atharva Tattu, Prajwal Chitode, Rushikesh Dhawne, Vedant Chaudhari	2024	The project combines generative AI with Command Line Interfaces (CLI) to simplify and automate command generation, addressing the complexities and steep learning curve associated with traditional CLI usage. By leveraging machine learning algorithms and natural language processing, AI models produce contextually relevant and syntactically accurate commands in real-time based on user input. [9]
Enhanced user interaction in operating systems through machine learning language models	Chenwei Zhang, Wenran Lu, Chunhe Ni, Hongbo Wang, Jiang Wu	2024	"This paper explores the potential applications of large language models, machine learning and interaction design for user interaction in recommendation systems and operating systems. By integrating these technologies, more intelligent and personalized services can be provided"[10]

- **Managing Linux servers with LLM-based AI agents:** This paper study usage of Large Language Model (LLM) to automate Linux server management tasks that needs good knowledge and big technical expertise. The study tested 150 tasks each in 9 different categories to execute tasks from file management to editing to program compilations. It measures and demonstrates the ability, efficiency and reliability of GPT-based AI agent to achieve and automate these complex tasks. This study introduces a new approach to manage Linux servers, showing the abilities of AI agents of handling complex tasks, making it easier and more automated. By adding a GPT-based AI agent with a Node.js server and a Dockerized Linux environment, the research demonstrated the AI gent's ability to autonomously interpret and handle 150 different server management tasks effectively. The results highlight how this approach reduces mistakes, human errors and enhancing efficiency and saving time. This research not only improves how servers are managed but also opens the door for using AI in other areas that involve complex tasks."[1]

  While this study focused on using GPT-based to automate 150 linux server tasks, our project used multiple open source models that run locally. Also our project provides web interface to deal with the system having multiple interfaces that serves the user. finally, in our project we followed a master-slave architecture were there is an agent who is responsible for distributing the tasks on other agents.

- **CLI Command Generation Using Generative AI:** The study demonstrates the integration of generative AI into Command Line Interfaces (CLI), showing its ability to simplify and automate command execution. The results highlight the AI's capacity to interpret user input, generate syntactically accurate commands, and automate repetitive tasks. Through real-time feedback, the AI responds to user needs to enhance usability and reduce errors. commands include file manipulation, script creation, and system management tasks, all completed efficiently with AI assistance.

  the project implemented a system where a Node.js-based application uses OpenAI's GPT-3.5 API to generate and execute commands. Examples of tasks given to the

AI include Listing files in a directory, Creating a Java file, Finding patterns in files, Creating an HTML file and Downloading packages.
The addition of generative AI into CLI environments represents a significant improvment in making command-line tools more intuitive, efficient, and user-friendly. The project showcases how AI can reduce the cognitive load on users, automate repetitive tasks, and enhance productivity in fields such as software development, system administration. Taking into consideration challenges such as ensuring accuracy, security, and usability. Overall, the work demonstrates the potential of AI-driven CLI tools to democratize access to powerful computational resources.[9]

While this project helps users generate and run commands in the terminal, our project goes further by fully automating system admin tasks. Instead of suggesting commands, our system assigns and executes tasks on its own. We also included tasks page where we can create specific tasks that are stored to be reused again, also our project provide monitoring and dashboard. Making our automation system bigger with multiple features.

- **Enhanced user interaction in operating systems through machine learning:** large language models and machine learning techniques offer great capabilities for user interaction and recommendation systems. They are set to elevate user experiences, improve recommendation accuracy, advance operating system development, and deliver more intelligent and personalized services.[10] By combining them more intelligent and personalized services will be returned, which improve user satisfaction. The rapid evolution of large language models opens new possibilities for innovative applications in recommendation systems and operating systems which make them ai-driven and develop new services , though it also raises challenges in data security and privacy protection.

3 System Design and Analysis

3.1 Preface

This chapter provides an in-depth exploration of the architectural and design decisions that form the foundation of our system. We delve into the intricacies of system components, data flow, and the technologies employed to create a robust system.

3.2 Requirements

3.2.1 Functional Requirements

1. **Tool Building**: Agents should have the ability to use tools to finish different tasks. Thus, Developing tools for using the terminal, opening directories and other operations will be the building blocks of these agents.

2. **Designing and Implementing Web Interface**: The user should have a centralized interface to access the system and interact with the different agents.

3. **Selective Agents**: The system will provide a feature to enable or disable the use of a selected agent by pressing the icon of the agent the user wants to enable/disable.

4. **Agents' Diversity**: The system composed of multiple agents, each handles OS Module like Networking, Services, Processes ...etc. Every agent should have suitable tools and knowledge in a specific field. No two agents should share the same capabilities, since the actions done by the agent should be only related to the domain they are assigned to. Thus, agents should cooperate when a task requires the use of multiple domains.

5. **Task Assignment**: The system should provide a screen for creating tasks and assigning them to agents.

3.2.2 Non-functional Requirements

1. **Reliability:** The system should be reliable and available 24/7, minimizing downtime and ensuring uninterrupted access for users.
2. **Reuse:** Tasks will be stored in the system which means we can re-execute the same tasks over and over as much we want.
3. **Performance** : The system should be able to fully utilize the CPU and GPU to efficiently run the models and deliver the needed task within a reasonable amount of time.
4. **Accuracy:** The system should should use the right agents to generate and execute command precisely.
5. **Maintainability:** The system should be built in a modular fashion in order to handle partition tolerance meaning if one agent stop working it doesn't affect the other agents.
6. **Usability:** The system should provide an intuitive and user-friendly interface that is easy to navigate. The system should provide clear and concise error messages that help users recover from errors.
7. **Security:** The system should be limited when it comes to executing system critical commands that may result in the system breaking down or deleting important files.

3.3 System Architecture

3.3.1 Agents Description

This section describes the AI Agents in the system and some of the functionalities and capabilities within each agent.

1. **Tasks Agent:** This Agent runs at the core of the system, it holds the responsibility of analyzing user prompt then generating a Plan (a data-structure used to represent which job belongs to which agent). In the Plan each agent is mapped to a job.

2. **Network Agent:**
    - ssh to remote server and execute commands.
    - list/enable/disable network interfaces.
    - start/kill http server in a directory.
    - list and connect to wifi networks.
3. **Users & Groups Agent:**
    - create/delete/modify users on the system.
    - create/delete/modify groups on the system.
    - manage permissions.
4. **Processes & Services Agent:**
    - start/stop/restart service on the system.
    - enable/disable/show status on the system.
    - kill processes & list logs of services.
    - run service in the background.
5. **Filesystem Agent:**
    - locate the path of a file or folder.
    - compress/decompress files on the system.
    - open files/folders/videos/images.
6. **Shell Agent:**
    - runs custom specific commands that aren't supported by other agents.

7. **Package Management Agent:**
   - install/remove/clean software packages on the system.
   - searches for specific package.
   - display package information and dependencies.
8. **Code Agent:**
   - Compile C/C++ files and run them.
   - write and run Bash Scripts and other programs.
   - creating cron jobs on the system.

3.3.2 Agentic Architecture

To develop an agent capable of performing the tasks of an operating system, we adopted the master-slave architecture to facilitate co-ordination between agents. Tasks Agent is the master it consumes the user prompt and analyzes it, afterwards this agent produces a data-structure called Plan, contains a list of Steps, where in the Step we have the name of an agent and the task assigned to that agent.

```
class Step(BaseModel):
    Agent: str = Field("Name of the agent that should execute this Step")
    Task: str = Field("The task assigned to the Agent")

class Plan(BaseModel):
    Steps: List[Step] = Field("List of Ordered Steps.")
```

Figure 6: Plan Data Structure illustration

When the Plan is generated we delegate each Step of the Plan to the right slave Agent the purpose of that agent is to complete the assigned task.

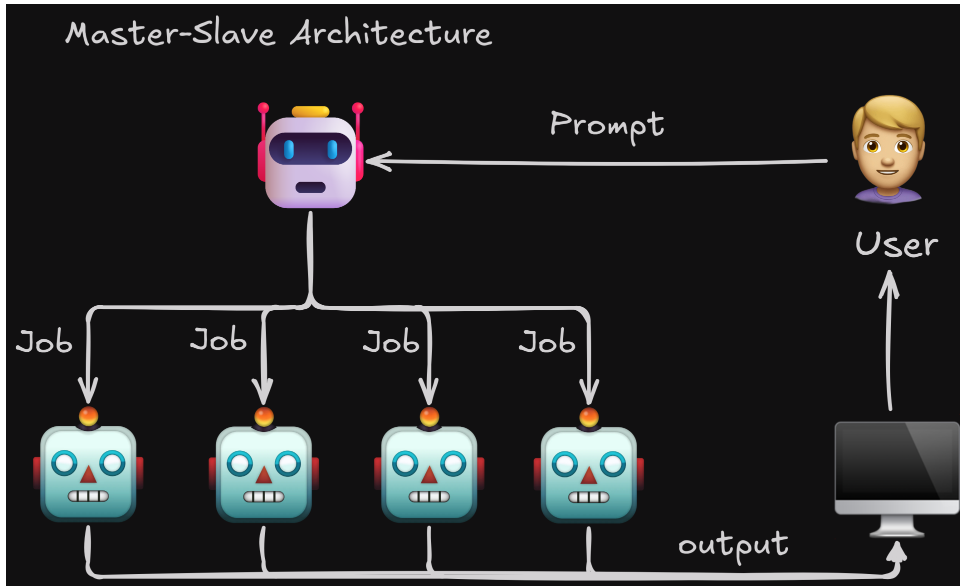


Figure 7: Master Slave Architecture

3.3.3 Workflow of Agents

The workflow the AI agents are going to follow is called the ReACT workflow which is designed to integrate the ability to reason and take action within the LLMs. ReACT involves a process of iterative reasoning and feedback.

Figure 8: ReACT Architecture

First step involves reasoning about what the user want. Second the LLM takes Action and executes a tool which results in an observation. we feedback this observation to the

AI Model again, the model then decides wether this output satisfies the user prompt if so the model exits the loop, other wise the model will choose to execute a different action.

3.3.4 Analysis and Diagrams

This section includes important illustrations of the system like the use case diagram and other sequence diagrams. These diagrams express the inner workings of the system. Figure 9 shows the use case diagram of our project.

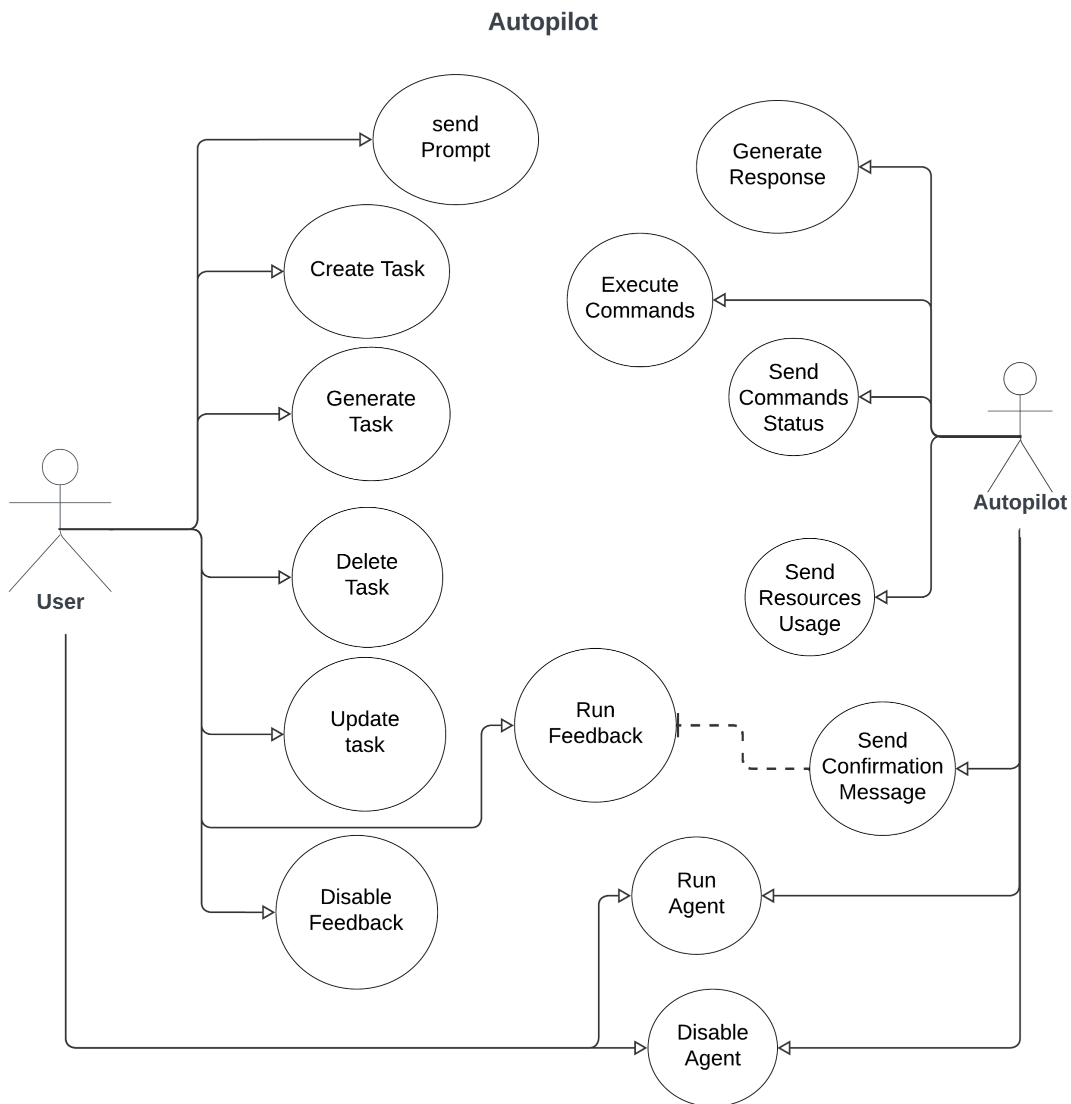


Figure 9: Use Case Diagram

Use Case	**Send Prompt**
Goal	Send prompt to the system to execute system commands.
Preconditions	• The related agents must be running. • The prompt must be clear.
Scenario	1. Run the related agents (press on the agents icon). 2. The user types some text in the input section and sends the prompt. 3. If the feedback is on, the system sends confirmation with the generated commands to run. The user can accept or reject. 4. If the feedback is off, the system will execute the generated commands without returning to the user.
Exceptions	• Commands outside the scope of the system. • The AI model is not running. • Agents are not running. • Connection issues.

Table 3: Use Case: Send Prompt

Use Case	**Create Task**
Goal	Create a task that contains a list of commands/jobs.
Preconditions	None.
Scenario	1. The user presses on the task creation button. 2. The task screen will pop up. 3. The user can add a command, edit the prompt, set the agent (change agent from the icon). 4. The user presses the Save button.
Exceptions	• Internal errors. • No agent chosen (agent icon not set).

Table 4: Use Case: Create Task

Use Case	**Generate Task**
Goal	To make the AI generate the task by entering a prompt. The generated task contains a list of generated commands, each with the agent and the generated prompt.
Preconditions	• The user must press the generate task button. • The user must enter a clear prompt.
Scenario	1. The user presses the generate task button. 2. A small popup screen appears. 3. The user writes a text about the desired task. 4. The user presses the generate button. 5. The system returns the generated task and refreshes the list of tasks on the user's screen.
Exceptions	• Unclear or ambiguous prompt. • Internal error.

Table 5: Use Case: Generate Task

Use Case	Delete Task
Goal	Delete a task from the tasks page.
Preconditions	• The user must select the desired task.
Scenario	1. The user selects a task. 2. The user presses the delete button. 3. The user confirms the deletion.
Exceptions	• Task not found.

Table 6: Use Case: Delete Task

Use Case	Update Task
Goal	To update task details, including changing the order of execution, editing a command's agent, or modifying the prompt.
Preconditions	• The user must double-click the desired task.
Scenario	1. The user double-clicks the task. 2. The user edits the task. 3. The user clicks the save button.
Exceptions	• Not clicking the save button. • Task not found. • Internal error.

Table 7: Use Case: Update Task

Use Case	Disable Feedback
Goal	To disable the execution confirmation message.
Preconditions	• Feedback must be On.
Scenario	1. The user presses the Feedback icon on the autopilot page.
Exceptions	• Internal error.

Table 8: Use Case: Disable Feedback

Use Case	Run Feedback
Goal	To enable the execution confirmation message.
Preconditions	• Feedback must be Off.
Scenario	1. The user presses the Feedback icon on the autopilot page.
Exceptions	• Internal error.

Table 9: Use Case: Run Feedback

Use Case	**Run Agent**
Goal	To enable an agent manually from the autopilot page.
Preconditions	• Agent must be off.
Scenario	1. The user presses the desired agent icon.
Exceptions	• Internal errors.

Table 10: Use Case: Run Agent

Use Case	**Disable Agent**
Goal	To disable an agent manually from the autopilot page.
Preconditions	• Agent must be running.
Scenario	1. The user presses the desired agent icon.
Exceptions	• Internal errors.

Table 11: Use Case: Disable Agent

Use Case	Generate Response
Goal	To make autopilot return a response to the user after commands execution.
Preconditions	• Commands execution must be completed.
Scenario	1. **If Autopilot Page:** The system uses an LLM to generate a list of commands and distribute them among agents. 2. **If Task Page:** The system retrieves the task containing a list of jobs with their agents from the database using the provided task ID. 3. The autopilot sequentially executes commands/jobs based on their order. 4. After completing the commands, the system returns a response indicating what has been done.
Exceptions	• Commands out of scope. • Errors in communication with the LLM (e.g., Ollama model). • Errors in generating responses from the LLM model.

Table 12: Use Case: Generate Response

Use Case	Execute Commands
Goal	To execute system admin commands using the shell.
Preconditions	• The user must send a prompt or task ID. • The LLM model must be running to enable communication.
Scenario	1. The system retrieves the list of jobs/commands. 2. Autopilot selects a command and uses an agent to execute it. 3. The agent writes the generated command to the shell and runs it.
Exceptions	• Execution errors. • Shell failures. • LLM failures.

Table 13: Use Case: Execute Commands

Use Case	Send Command/Job Status
Goal	inform user about each command's status (running, completed, or failed).
Preconditions	• A websocket connection called ”notifications” must be initialized. • The frontend must be connected to the websocket.
Scenario	1. After a command is finished, autopilot backend sends a message to ”notifications” websocket. 2. The message contains the command's status. 3. The frontend receives the message and updates the UI based on the command's status.
Exceptions	• Communication error.

Table 14: Use Case: Send Command/Job Status

Use Case	**Send Resource Usage**
Goal	To send information to the user about CPU, RAM, GPU, and GPU memory usage.
Preconditions	• A websocket connection called ”monitor” must be initialized and connected.
Scenario	1. The user presses the Dashboard page. 2. The frontend connects with the ”monitor” websocket. 3. The frontend receives information every second about usage percentages. 4. A React.js chart library represents this information on the screen.
Exceptions	• Communication error.

Table 15: Use Case: Send Resource Usage

Use Case	**Send Confirmation Message**
Goal	To show the end-user the commands about to be executed and allow them to accept or reject.
Preconditions	• Feedback agent must be on. • A websocket connection called ”tools” must be set.
Scenario	1. The user sends a prompt to the chat page. 2. The backend sends a confirmation message to the end-user. 3. small pop-up message appears with the commands to be executed. 4. The user can accept or reject the execution.
Exceptions	• Communication errors.

Table 16: Use Case: Send Confirmation Message

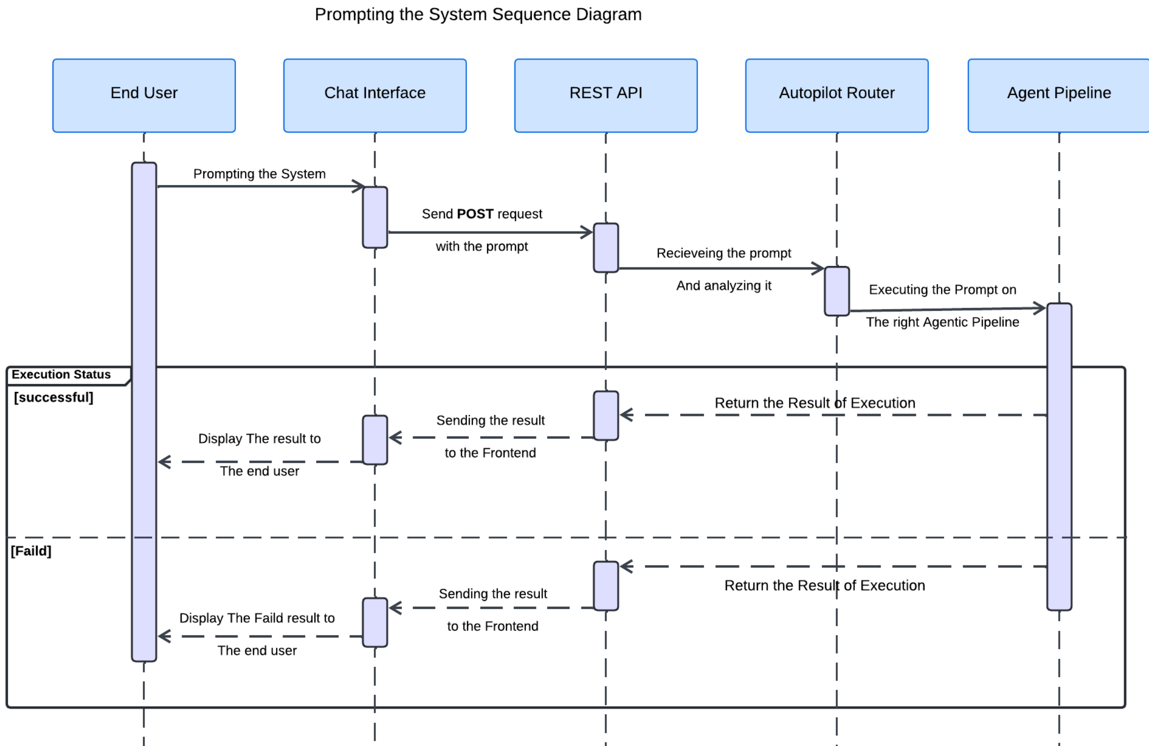


Figure 10: Prompting Sequence Diagram

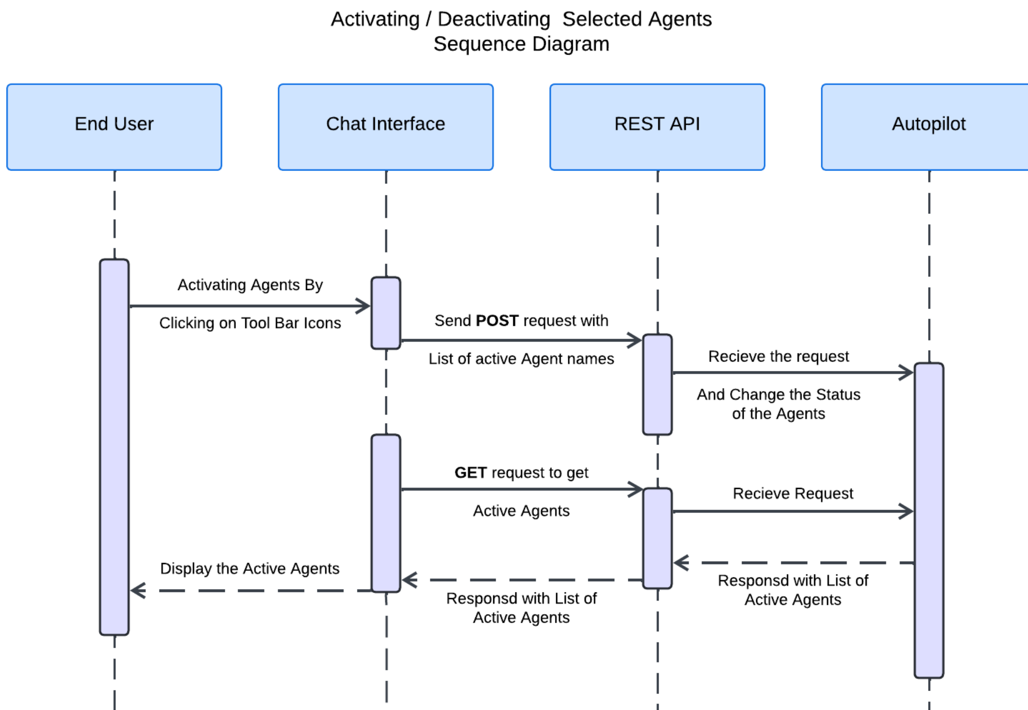


Figure 11: Agent Activation Sequence Diagram

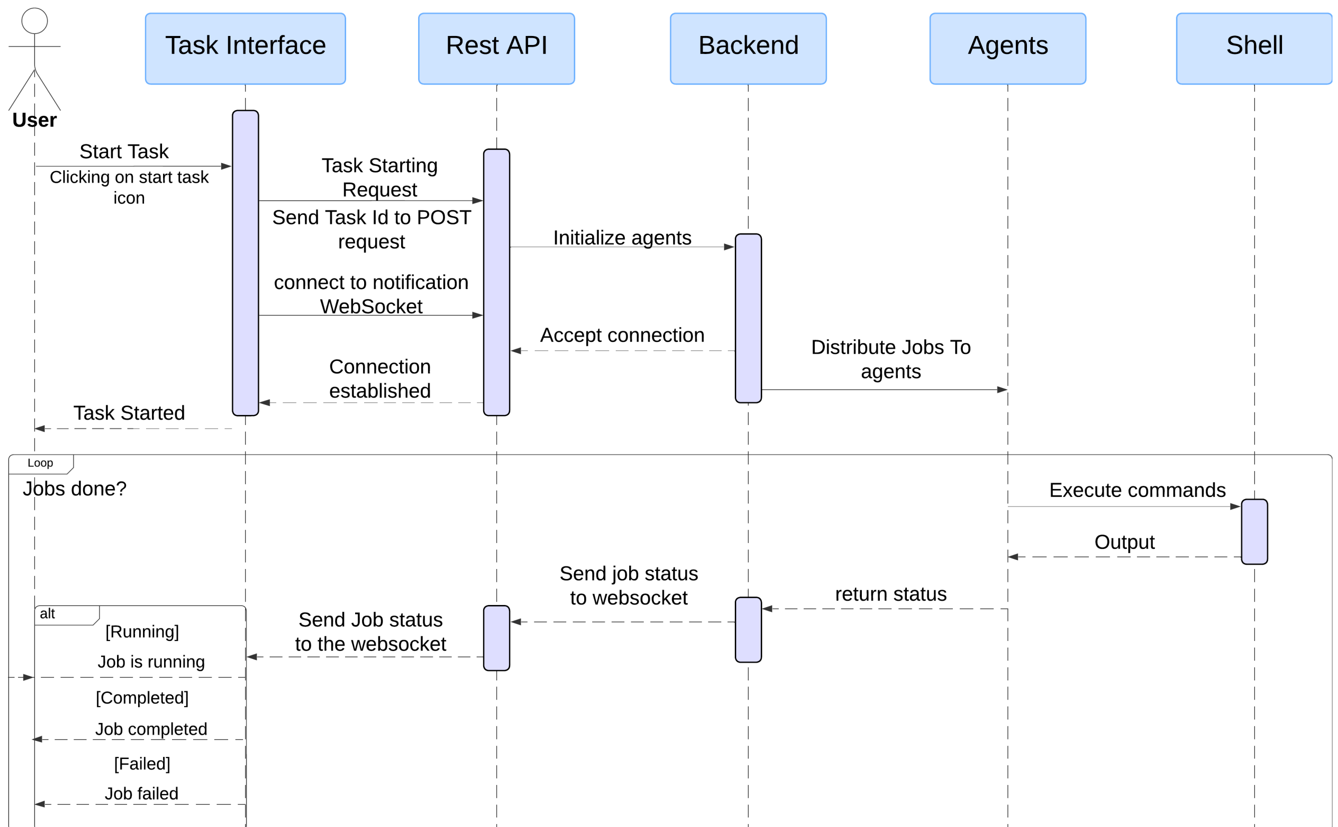


Figure 12: Task Execution Sequence Diagram

3.4 Technologies and Frameworks

This section displays the technologies and frameworks that will serve the process of building the system.

1. **Python :** is the primary language that will be used to develop the system the main reason is the flexibility along with the convenience provided by the libraries it has which targets developing AI Agents.

2. **Fastapi :** is a python library that enables the developer to quickly build REST API's to focus more on more important backend operations.

3. **Ollama :** is an open-source AI models provider that runs in the terminal, you can think of it as an online hub that contains open-source AI models that you use to pull down to your local machine some LLMs and then run them. On top of that ollama provide python libraries to enable the integration of LLMs with the python code you write.

4. **LangChain :** is a python library that assists the python developer in the process of integrating tools with LLMs and enhancing the LLMs response by using prompt templates. In addition to that langchain support the concept of chaining different components together and executing them as a single chain.[4]

5. **LangGraph :** is a python library build by langchain team the purpose of langgraph is be able to engineer or design the process or the workflow of the agent by representing it as a graph with nodes and edges. Whats great about this is that if we draw a flow chart of how the agent should act we can reflect it in code by building a graph.

6. **Redis Database :** is in-memory key-value database, It is designed for high performance, offering low-latency operations. In our system Tasks, Jobs and Active agents are going to be stored in the redis database.

7. **React JS :** The frontend is developed using ReactJs, a powerful Javascript framework for building dynamic web applications and handling all user interactions and providing a responsive user interface.

4 Implementation and Evaluation

4.1 Handling Alternatives

This section explores technologies and alternative for building this project.

Subject	**Alternatives**	**Reason**
Database	Redis vs. SQL Databases	We need to store simple Tasks as data structure with fast read/write operations. so we use Redis key-value database.
Backend	FastAPI vs. Django	FastAPI offers building endpoints quickly allowing developer to focus more on the core of the project.
Operating System	Linux vs. Windows	Most sys-admins use Linux, and for it's simplicity to automate the OS using scripts and programs.
AI Models	Open vs. Closed Source	Open-Source models are suitable since we need to run them locally on the same machine the system works on.
AI Frameworks	LangGraph vs. CrewAI	LangGraph allows us to control the details of how agent behaves in-addition to flexibility of troubleshooting issues.
Architecture	Master-Slave vs. Single Agent	One Agent handling different modules can lead to LLM hallucinations, to avoid them we distribute modules to different AI Agents.
Language	Python vs. Other Languages	Python encompasses a large number of libraries for integrating LLM's into applications and automating systems.

Table 17: Handling Alternatives

4.2 Building The Agents

4.2.1 AI Models Used

1. **Llama3.2** : is an open-source AI Model release by Meta, It's used in this project inside the AI Agents to select or choose the right command or tool to be executed based on the given task.

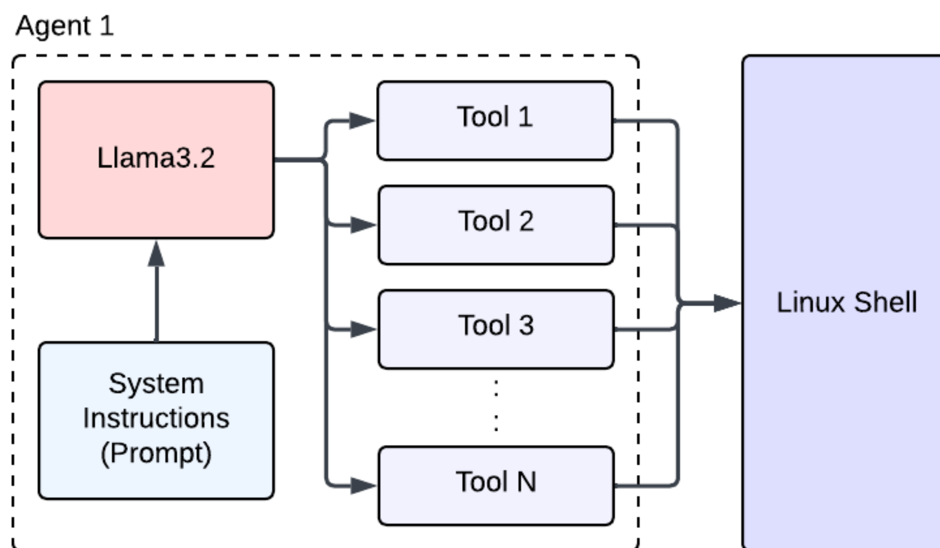


Figure 13: Llama3.2 Agent Integration

2. **LinuxGeneral** : is an fine-tuned Llama3.1 model on Linux dataset, can be downloaded from HuggingFace coummunity. It's used to suggest a list of commands to the Llama3.2 model to be executed on the system.

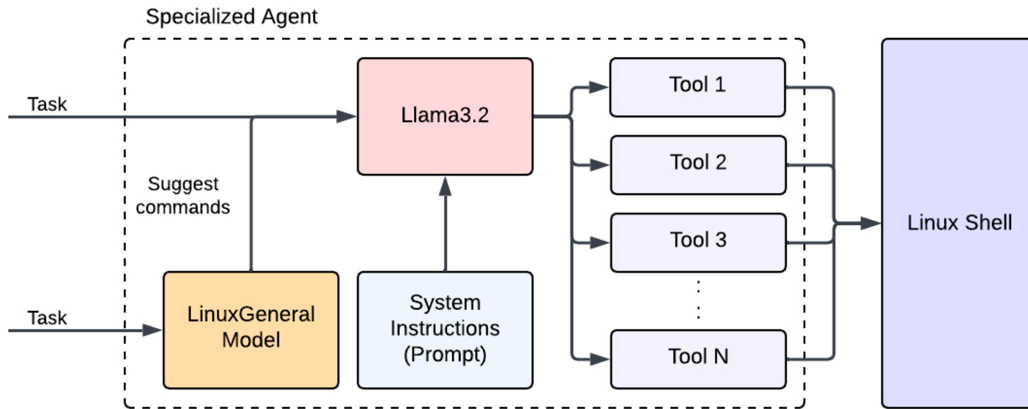


Figure 14: LinuxGeneral Agent Integration

3. **Llama3.1** : is an open-source AI Model released by Meta, in this project we use it when we need to generate Tasks for the user based on a prompt, or when we need to send task to the right agent.

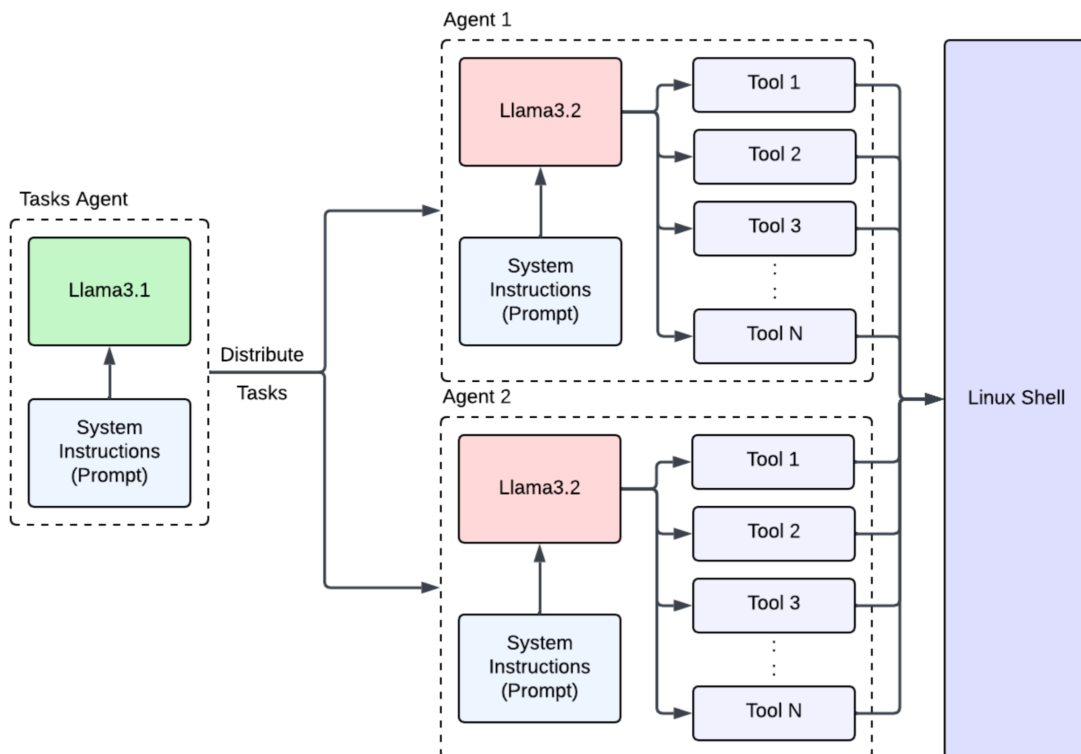


Figure 15: Llama3.1 Agent Integration

4. **Deepseek R1** : is an open-source AI Model released by DeepSeek, in this project after all the agents finsih executing Jobs we combine the output of these jobs, and ask this to model to summarize it.

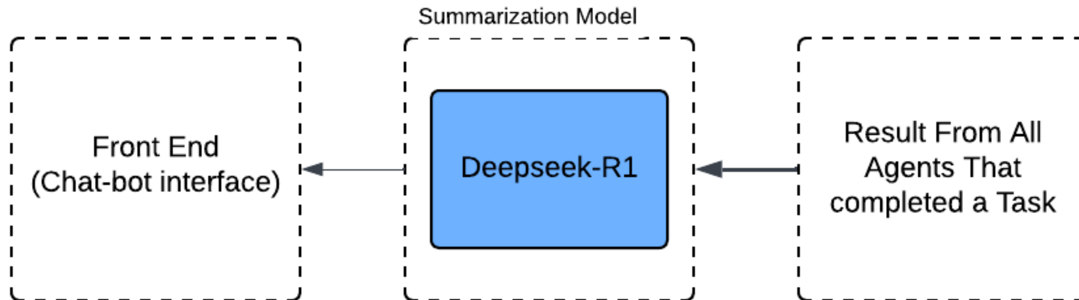


Figure 16: DeepSeek-R1 Integration

4.2.2 Models Specification

Model	**Specification**	**Provider**
Llama3.2	2 billion parameters runs locally with 2 Gigabytes of size.	Ollama
LinuxGeneral	8 billion parameters runs locally with 17 Gigabytes of size.	HuggingFace
Llama3.1	8 billion parameters runs locally with 4.7 Gigabytes of size.	Ollama
Deepseek-R1	7 billion parameters runs locally with 4.7 Gigabytes of size.	Ollama

Table 18: Models Specification

4.3 Building The Tools

4.3.1 Creating Tools

A tool represents a python function that LLM can executes if it meet the user task. the following is an example of a tool used to ssh to a remote server and run commands on that remote server, the tool is split into the code and the comments before it, these comments are read by the LLM learn when it's suitable to execute this function.

```
 6 def ssh_to_host(username: str, hostname: str, password: str, commands: list[str]):
 7     """
 8         ssh_to_host is a tool that connectes and executes
 9         commands to remote host or server.
10         it takes in a list of commands
11         remember the password is a str
12         Args:
13             username: str
14             hostname: str
15             password: str
16             commands: list of str
17         Returns:
18             the output of executing commands on remote
19             server
20     """
21     output = ""
22     try:
23         session = pxssh.pxssh()
24         session.login(hostname,username,password)
25         start_terminal(f"ssh root@{hostname}")
26 
27         for command in commands:
28             session.sendline(command)
29             session.prompt()
30             template = f"[{username}@{hostname}]$ {command}"
31             send_to_terminal(template)
32             tmp = str(session.before.decode()).split('\n')[1]      ▪ "decode" is no
33             send_to_terminal(tmp)
34             output += template + "\n" + tmp
35         session.logout()
36     except Exception as e:
37         output += f"something went wrong {e}"
38     finally:
39         return output
40
```

Figure 17: Tool Code Sample

After creating a set of Tools (functions) that automate running tasks on the system, we collect similar Tools together in a toolkit and in our case we create a network toolkit, and Finally we create a model and bind that model with the toolkit to create our agent.

```
147 def get_network_toolkit():
148     return [list_wifi_networks,
149             list_interfaces,
150             list_network_hosts,
151             enable_interface,
152             disable_interface,
153             start_http_server,
154             kill_http_server,
155             ssh_to_host]
156
157 tools = get_network_toolkit()
158 llm = ChatOllama(model="llama3.2", temperature=0)
159 agent = llm.bind_tools(tools)
```

Figure 18: Binding Tools With LLM

4.4 Tasks & Jobs Feature

4.4.1 Creating Tasks & Jobs

Tasks feature allows the system administrator to create Tasks, inside each Task you can create a list of Jobs, each Job consisting of what you want to execute on the system in natural language and the AI Agent responsible for doing this Job.

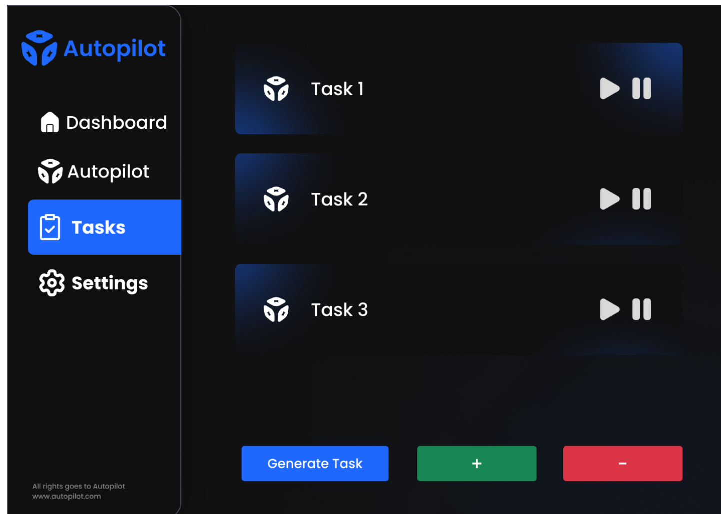


Figure 19: Tasks

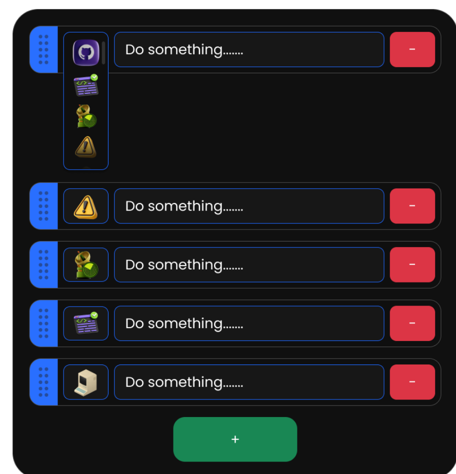


Figure 20: Jobs

There are two ways to create Tasks:

1. Manual: The user manually creates a task and manually assigns it to each agent, requiring careful attention to match the task with the right agent based on their abilities. This relies heavily on the user's understanding of each agent's capabilities.

2. AI Generated: the user clicks on the Generate Task button and writes a prompt of what exactly he needs to be done, and in the background an LLM analyzes the prompt and automatically creates a Task with a list of jobs for each agent.

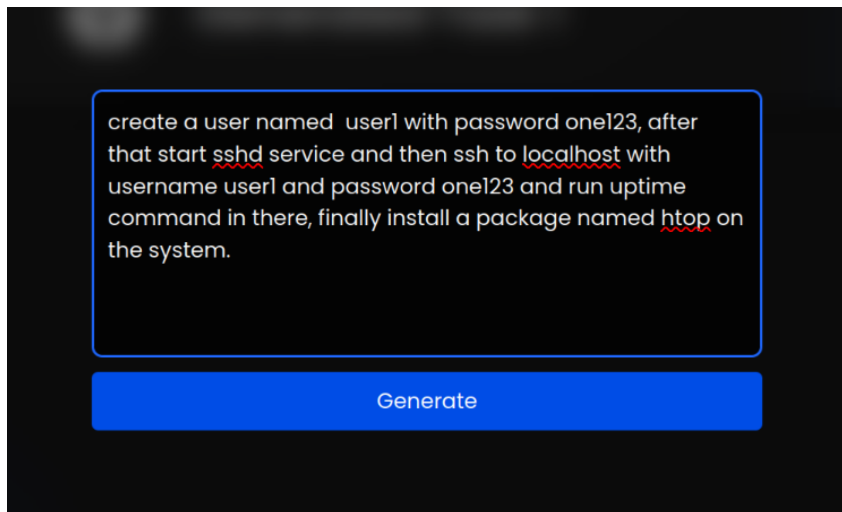


Figure 21: Generating Task From Prompt

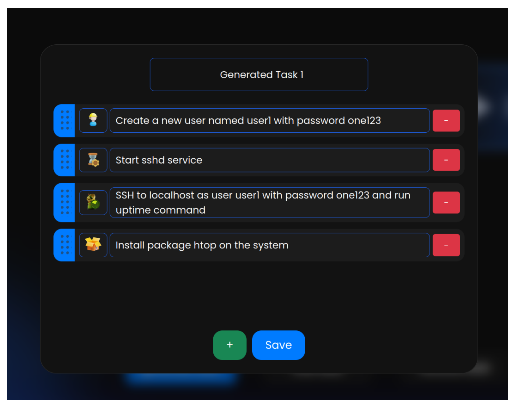


Figure 22: Generated Jobs

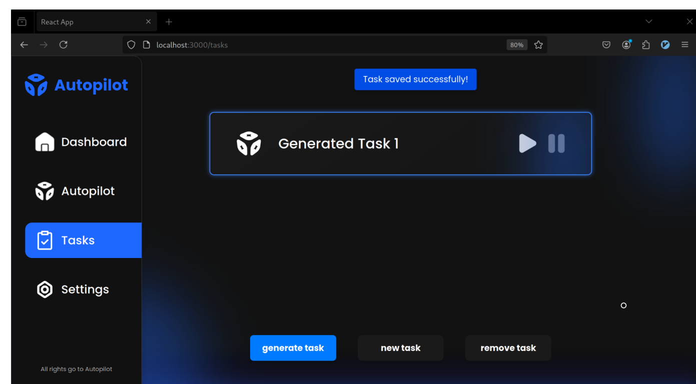


Figure 23: Generated Task

4.4.2 Running Tasks & Jobs

When the user presses start on a Task, each Job is sequentially executed by the suitable AI Agent.(see Figure 12 task execution sequence) In addition to that a Terminal pops up and displays each command executed by the agents.

Every Jobs passes through 3 main states:

1. **Job Running**: it's marked by the color blue and it shows which jobs is currently being executed.

2. **Job Completed**: it's marked by the color green, which indicate that the job was finished and was ran successfully without any issues.

3. **Job Failed**: it's marked by the color red, which indicates that the job failed.

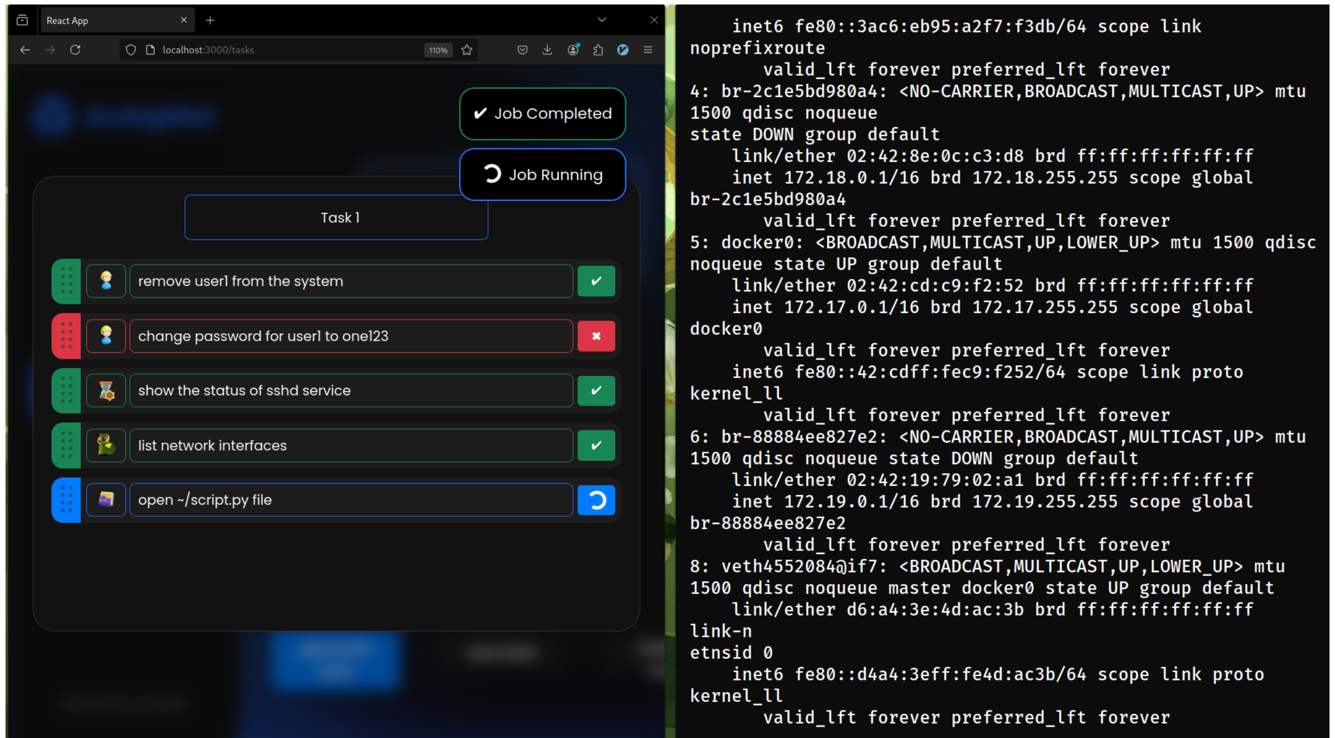


Figure 24: Executing Jobs

4.4.3 Tasks API Endpoints

Endpoint	Description
GET /tasks	Retrieves a list of all tasks.
GET /tasks/{id}	Retrieves a single task.
DELETE /tasks/{id}	Deletes a task with id.
POST /tasks	Creates a new task in the redis server.
POST /generate-task	Generates a task using LLM based on prompt.
POST /tasks/{id}/start	Invokes the agents to start running tasks.
POST /tasks/{id}/stop	Interrupts running agents to stop a task.
WS /notification	Websocket tells frontend if job is Running, Failed or Done.

Table 19: Tasks API Endpoints

4.5 Dashboard Feature

Dashboard Screen is important for two main things, the first is monitoring the system resource usage and learning information about the AI Agents.

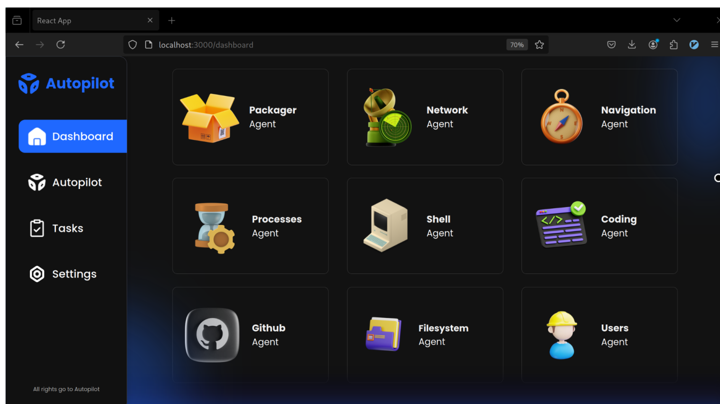
4.5.1 Monitoring Screen

Monitoring Dashboard is used to display the usage of system resources by the AI Models running on the system. since we use open-source AI models that run locally on the system there is going to be a heavy GPU utilization, monitoring this usage in addition to monitoring overall system usage is an essential feature.

Figure 25: Monitoring Screen

4.5.2 Agents Dashboard

The system also provides Agents Dashboard that contains a list of cards representing the Agents system supports and the description and tools each agents is capable of using. The following Images showcase the Agents Dashboard, by clicking on each card a pop-up with a description of the Agent shows up to explain what that Agent does.



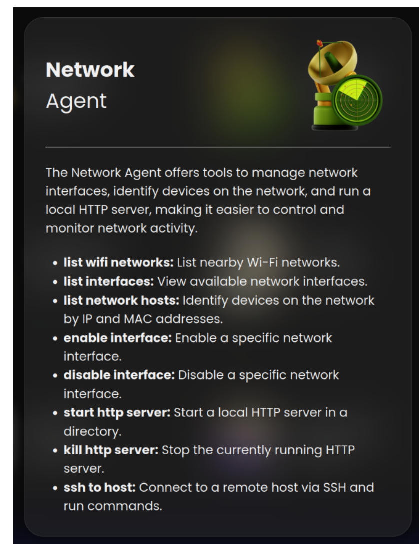


Figure 26: Agents Dashboard

Figure 27: Agent Description

4.5.3 Dashboard API Endpoints

The description of each agent in stored in the frontend inside local storage of the browser, while we utilized a web-socket that submits real-time data about system resource utilization to the frontend.

Endpoint	**Description**
WS /monitor	websocket used that gets updates on CPU, GPU Usage, GPU Memory Usage, RAM, .

Table 20: Dashboard API Endpoints

4.6 Chatbot Feature

4.6.1 Interfaces & Descriptions

Another major feature that the system supports is the Chat-Bot Feature. This feature serves three purposes:

1. **Executing Quick Tasks:** if there is a need to execute a quick task on the OS without the need to create a task and start mapping each task to a specific agent, the Chat-Bot feature enables quick task execution. (Figure 10: prompting sequence diagram clarify the process)

2. **Summarization:** after executing some task, there is a need to summarize what happened on the OS. This feature generates a summary after each task is executed. (DeepSeek r1 is responsible for this as shown in Figure 16: DeepSeek-R1 Integration)

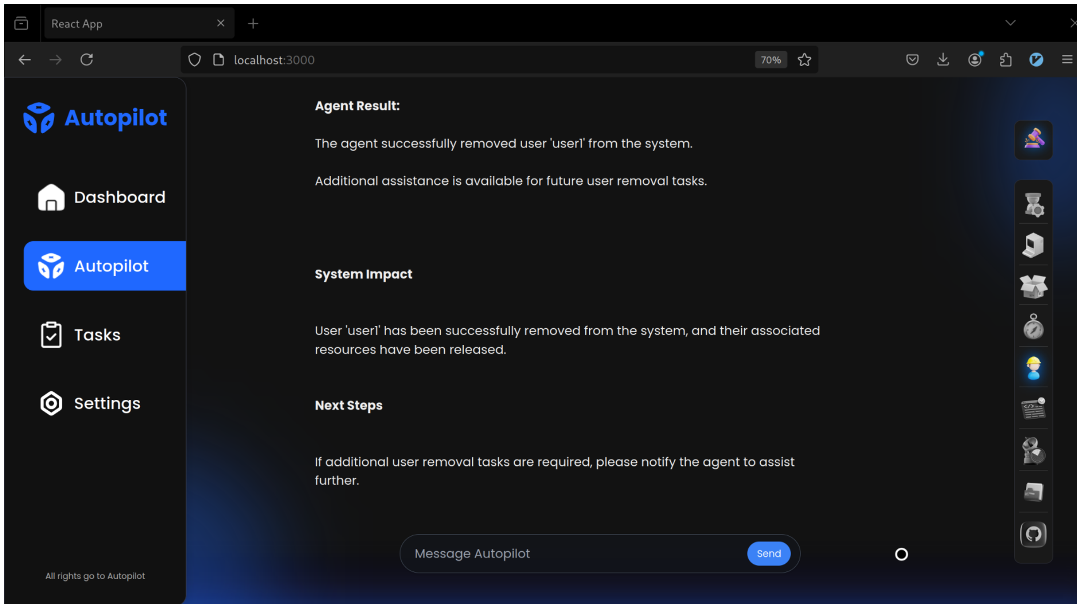


Figure 28: Chat-Bot Summarization

3. **Enabling Feedback:** The Chat-Bot has a feature called feedback, this feature show the end-user what command is about to get executed and based on that the user can decide to accept executing that command or reject it.
(enabling/disabling feedback is the same as enabling/disabling agents in Figure 11: Agent Activation Sequence Diagram)

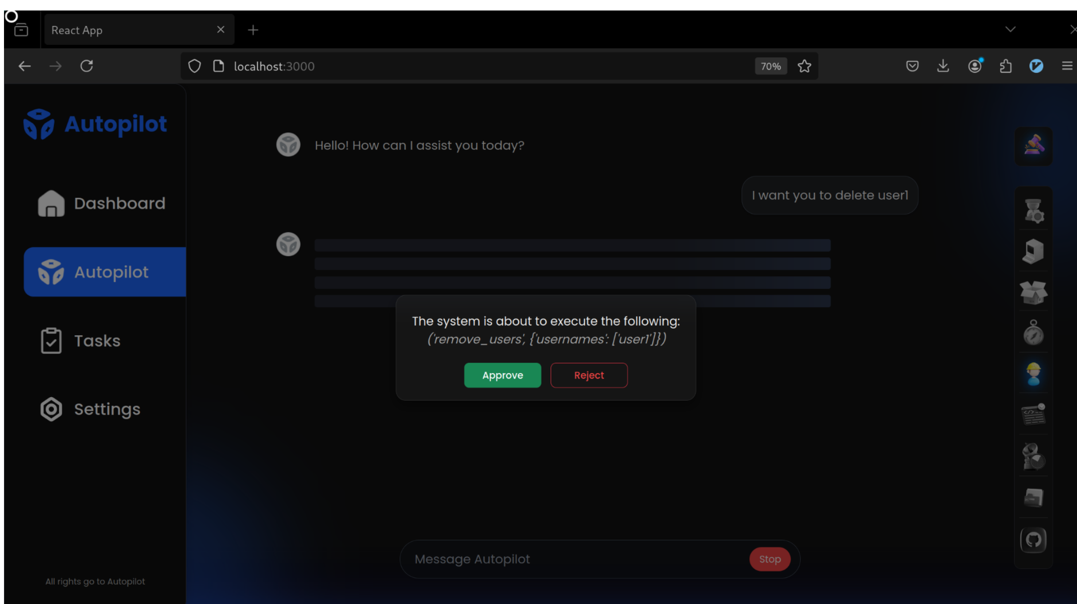


Figure 29: Taking Approval From User

4.6.2 Chat-Bot API Endpoints

Endpoint	Description
GET /toolbar	Retrieves a list of Active Agents.
GET /feedback	Returns True if feedback is enabled else false.
POST /toolbar	Activate or Deactivate Agents.
POST /feedback	For Enabling or Disabling Feedback.
POST /accept	Sends Approve for executing a command to the backend.
POST /reject	Sends Reject for executing a command to the backend.
POST /chat	Chat or Send a Quick Task to the Chat-Bot.
WS /tools	Web-socket used to notify the frontend when the system is about to run a tool and take user approval.

Table 21: Chat-Bot API Endpoints

4.7 Auto-Correction Feature

The result of executing a task can lead to a successful or failed task, in the case of a failed task the end-user can tackle this issue by enhancing the prompt and illustrate more clearly what he needs. In order to decrease the need for user intervention we made the agents reflect on the errors. After finding a failed task we resend the same task to the same agent until the task returns a successful result or the system bypasses the limit we put for re-doing the task, this is important since we don't want the system to go into infinite loop.

```
executor = AgentExecutor(
    agent=llm,
    tools=tools,
    verbose=True,
    max_iterations=2,
)
```

Figure 30: Max Iterations

In Figure 30 we can see that when we define an Agent in langchain it gives us the ability to retry the execution using the **max_iterations** parameter. In our case it's set to two, which means it will execute only 2 times if the task still fails manual intervention is required from the end users.

4.8 Evaluation

4.8.1 Evaluating Modules

In this section we evaluate the performance of the system against test cases in different aspects of OS modules in each table we have the name of the task followed by status of wether the task failed, succeeded or how many tries it took for it succeed. (✓) represents success and (✗) represents failure (✗✗✓) represents that it failed the first two times and was successful the last time.

Network Service	Task	Status
1.1	Enable and Disable network interfaces	✓
1.2	List network interfaces and wifi networks	✓
1.3	ssh to remote host and execute commands	✗✗✓
1.4	start or kill http server	✓

Table 22: Network Tasks Evaluation Table

Users & Groups	Task	Status
2.1	create multiple users with random passwords	✗✓
2.2	change password for userX from X to Y	✓
2.3	add multiple users to a group	✗✓
2.4	create groups on the system	✓
2.5	delete groups on the system	✓

Table 23: Users & Groups Tasks Evaluation Table

Software Packages	Task	Status
3.1	Install or Remove List of software packages at once	✗✓
3.2	Search for a specific package in repository	✓
3.3	Update system and system packages	✓
3.4	Clean installed packages cache	✓
3.5	List dependencies on the system	✓

Table 24: Software Packages Evaluation Table

Software Development	Task	Status
4.1	Compile C/C++ file and run it.	✓
4.2	Running Python scripts and displaying the output	✓
4.3	Running bash scripts and displaying the output	✓
4.4	Create, remove and modify github repositories	✗✓

Table 25: Software Development Evaluation Table

Filesystem Management	Task	Status
5.1	open text files, folders, images and videos.	✓
5.2	compress and decompress a file.	✓
5.3	find location or path of a file or folder	✓

Table 26: Filesystem Evaluation Table

Processes and Services	Task	Status
6.1	List system processes.	✓
6.2	Kill a process by name.	✓
6.3	Start, stop, restart services and daemons.	✓
6.4	Enable, disable, reload services and daemons.	✓
6.5	Run command in the background.	✗✗✓
6.6	Get logs of a running service.	✓

Table 27: Processes & Services Evaluation Table

4.8.2 Evaluation Statistics

Evaluated Module	Success Rate	Failure Rate	Overall Success	Overall Failure
Network Management	83.333%	16.666%	11.42%	5.71%
User & Group Management	71.42%	28.57%	14.28%	5.71%
Software Development	80%	20%	11.42%	2.85%
Package Management	83.333%	16.666%	14.28%	2.85%
Processes & Services	75%	25%	17.14%	5.71%
File-system Management	100%	0.0%	8.57%	0.0%
Average Rates	98.61%	1.39%	—	—

Table 28: Success & Failure Rates Table

4.9 Benchmarking

This section includes the benchmarks for cpu and gpu usage and tempreture. In addition to gpu memory usage and the time it took for executing the task.

4.9.1 Methodology

When /tasks/id/start endpoint is invoked we retrieve the task from memory and loop over all jobs in the task to execute them. In order to benchmark our system we record the time before executing then record the time after executing the task and take the difference between them, after that we created benchmark_system() function that records cpu and gpu metric finally we write the time and metrics to benchmark.csv file using write_benchmarks() function.

```
249 @app.post("/tasks/{id}/start")
250 async def start_task(id: int):
251     try:
252         data = memory.get(f"task:{id}")
253         jobs = json.loads(data)
254 
255         for job in jobs['commands']:
256 
257             before = time.time()
258 
259             result += await process_job(job, socket)
260 
261             after = time.time() - before
262             metrics = await benchmark_system()
263             await write_benchmarks(metrics , job['task'], after)
264 
265 
```

Figure 31: benchmark the system

The following is the code for benchmarking the system, we use psutil and GPUtil libraries to capture information about usage, memory, and temperature, then return them in a dictionary.

```python
async def benchmark_system():
    metrics = {}
    cpu_usage = psutil.cpu_percent(interval=1)
    metrics['cpu_usage'] = cpu_usage

    gpus = GPUtil.getGPUs()
    gpu = gpus[0]
    metrics['gpu_usage'] = gpu.load * 100
    metrics['gpu_memory_used'] = gpu.memoryUsed
    metrics['gpu_temp'] = gpu.temperature

    temps = psutil.sensors_temperatures()
    metrics['cpu_temp'] = temps['coretemp'][0].current

    return metrics

```

Figure 32: benchmark_system() function

The following picture holds the code for writing the benchmarks to a csv file.

```python
async def write_benchmarks(metrics, task, after):
    row = {
        'task': task,
        'cpu_usage': metrics['cpu_usage'] * 10,
        'cpu_temp': metrics['cpu_temp'],
        'gpu_usage': metrics['gpu_usage'] * 10,
        'gpu_memory': metrics['gpu_memo'],
        'gpu_temp': metrics['gpu_temp'],
        'time': after
    }

    headers = ['task', 'cpu_usage', 'gpu_usage','gpu_memory', 'cpu_temp', 'gpu_temp', 'time']
    with open("benchmark.csv", mode='a', newline='') as file:
        writer = csv.DictWriter(file, fieldnames=headers)
        if file.tell() == 0:
            writer.writeheader()
        writer.writerow(row)

```

Figure 33: write_benchmark() function

4.9.2 Results

The following table summarizes the results from the benchmark.

Prompt	CPU Usage (%)	GPU Usage (%)	GPU Memory (MB)	CPU Temp (°C)	GPU Temp (°C)	Time (s)
Install htop software package	48	80	5164	73	67	20.20
Remove neofetch software package	46	90	5103	75	71	2.68
Clear the cache of the packages	46	90	5098	79	73	5.39
Update the system	43	80	5079	78	72	25.14
Run the following Python file `/home/ha1st/run_me.py`	52	80	5080	80	76	2.11
Run this Bash script `/home/ha1st/run_me.sh`	47	80	5078	81	77	2.16
Compile the following file `/home/ha1st/run_me.cpp`	52	100	5084	84	76	2.39
Open `~/Videos` directory	96	100	5133	85	79	2.19
Open `~/run_me.py` file	100	100	5140	93	79	3.85
Create a group named `devs`	100	80	5139	93	80	2.93
Create a user named `developer1` with password `123one`	41	80	5093	85	76	7.75
Change the password for `developer1` to `one123`	43	90	5092	81	75	5.17
List network interfaces	59	80	5101	83	78	2.67
Add `developer1` to `devs` group	48	100	5095	85	79	3.18
Delete `devs` group	42	90	5099	85	81	2.89
Delete the user named `developer1`	39	90	5102	93	82	3.03
Start `sshd` services	38	80	5101	88	83	2.65
Stop `sshd` service	41	90	5098	87	83	2.66
List processes on the system	41	90	5142	90	80	4.34
Start HTTP server inside `/home` directory	43	80	5150	91	84	2.32
Kill HTTP server	46	80	5146	88	84	2.14

Table 29: System performance metrics for different tasks

Important notes to consider is that some tasks like updating and installing a software package took a lot of time but didn't take high resource utilization unlike other tasks, the reason for that is some tasks like downloading packages from the internet adds network latency which increases the execution time. In addition to that we can deduce that our system needs to utilize the gpu more than the cpu that's because the model provider we use (ollama) uses cuda-toolkit and cuda drivers to run the models on the gpu to maximize the performance.

5 Conclusion

5.1 Summary

In this project, we have developed a comprehensive Agentic System that seamlessly automates running commands and tasks on a Linux Server. The system facilitates the power of open source models and Generative AI to translate Natural Language into Linux Shell commands to satisfy the end-users tasks while ensuring a rich user experience. The performance evaluation confirmed the system's ability to complete complex tasks in various OS management aspects. Overall, this project showcases our ability to make use of Generative AI for boosting the productivity of system adminstrators.

5.2 Future Work

5.2.1 User Analytics Dashboard

Objective: Provide administrators board with insights into process behavior, network traffic, and system connections through detailed analytics.

Approach: Develop an analytics dashboard that visualizes data such as user engagement, search patterns, and metrics.

5.2.2 Logs System

Objective: Provide a Logs board with real-time logs comming from the system processes, services and network traffic.

Approach: Develop a feature that gather logs from system-wide files, make it in readable format and order them by time.

5.2.3 Task Scheduling

Objective: Provide the option to schedule tasks to run at a specific time.

Approach: Store additional data about scheduled time inside the Task entity and start

a background service that checks if its time to execute some task if so, then the task will be executed.

References

[1] C. Cao, F. Wang, L. Lindley, and Z. Wang. Managing linux servers with llm-based ai agents: An empirical evaluation with gpt-4. *Machine Learning with Applications*, 17(100570), Sept. 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S266682702400046X.

[2] Elastic. What are large language models (llms)? https://www.elastic.co/what-is/large-language-models, 2024. Accessed: 2024-12-11.

[3] GeeksforGeeks. What is linux operating system. GeeksforGeeks, Jul. 2024. Accessed: Dec. 10, 2024.

[4] LangChain. Chains documentation. [Online]. Available: https://python.langchain.com/v0.1/docs/modules/chains/. [Accessed: Aug., 2024].

[5] A. Ng. What's next for ai agentic workflows. Sequoia Capital, YouTube, Mar. 2024. [Online]. Available: https://www.youtube.com/watch?v=sal78ACtGTc.

[6] Pragna. Linux architecture. Medium, Aug. 2023. Accessed: Dec. 10, 2024.

[7] Cole Stryker and Jim Holdsworth. What is nlp (natural language processing)? IBM, Aug. 2024. Accessed: Dec. 10, 2024.

[8] SuperAnnotate. Llm agents: The ultimate guide. https://www.superannotate.com/blog/llm-agents, October 2024. Accessed: 2024-12-11.

[9] Atharva Tattu, Prajwal Chitode, Rushikesh Dhawne, and Vedant Chaudhari. Cli command generation using generative ai. Project Report, 2024. Submitted in partial fulfillment of the requirements for the Degree of Bachelor of Engineering in Computer Science and Engineering.

[10] C. Zhang, W. Lu, C. Ni, H. Wang, and J. Wu. Enhanced user interaction in operating systems through machine learning language models. In *Conference Proceedings of*

SPIE, volume 13180, June 2024. [Online]. Available: https://doi.org/10.1117/12.3033610.