A Report on

LLM OS: Using an AI-Powered Terminal for Code Generation and Execution

Submitted in partial fulfillment for the award of the degree of

BACHELOR OF TECHNOLOGY

in

INFORMATION TECHNOLOGY

by

Akhil Menon 231AI005
Dishanth Arya 231AI008
Krisha Arora 231AI015
IV Sem B.Tech (AI)

Department of Information Technology

National Institute of Technology Karnataka, Surathkal.

April 2025

CERTIFICATE

This is to certify that the report entitled "LLM OS: Using an AI-Powered Terminal for Code Generation and Execution" has been presented by *Akhil Menon (231AI005), Dishanth Arya (231AI008) and Krisha Arora (231AI015),* students of IV semester B.Tech (AI), Department of Information Technology, National Institute of Technology Karnataka, Surathkal, during the even semester of the academic year 2024 – 2025. It is submitted to the Department in partial fulfillment of the requirements for the award of the degree of Bachelor of Technology in Information Technology.

Place: _____

Date: (Signature of the Examiner1)

Place: _____

Date: (Signature of the Examiner2)

Place: _____

Date: (Signature of the Coordinator)

DECLARATION BY THE STUDENT

We hereby declare that the Course project report for IT371 entitled "LLM OS: Using an AI-Powered Terminal for Code Generation and Execution" was carried out by me during the even semester of the academic year 2024 – 2025 and submitted to the department of IT, in partial fulfillment of the requirements for the award of the Degree of Bachelor of Technology in the department of Information Technology, is a bonafide report of the work carried out by me. The material contained in this seminar report has not been submitted to any University or Institution for the award of any degree.

Place: _____

Date: _____

_____

(Names and Signatures of the Students)

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

An operating system is simply a system software, which acts as an interface between the user and the hardware. We know that to create an operating system, many components are necessary. A large language module (LLM) on the other hand is an artificial intelligence module that uses the concept of machine learning to understand and generate texts. In order to operate a proper LLM, we first need to train it on a large dataset. This enables them to perform tasks such as Natural Language Processing (NLP) techniques such as text generation and translation. Now, what if we combine the two concepts. We get a large language module operating system. An LLM OS is an experimental operating system that integrates large language modules into its core functionalities. By doing this, we can enhance the user interaction, automation and system management. The whole purpose of this experiment is to create an operating system where the large language modules. We can prompt this LLM OS to do a specific task and it will perform it. With proper integration, we can perform various tasks like command execution, workflow automation, system optimization, and user interaction.

An example where this may come in handy is the following. Suppose we want to open a web browser from the command terminal with an LLM OS, then we can simply prompt the LLM OS (on the terminal) to open the browser by giving the command “open browser”. After looking at the command, the LLM OS then has to open the browser for the user. Similarly, we can perform various tasks just by giving various prompts into the prompt command, and accordingly, the LLM OS will perform its work.

A smart terminal helper makes creating scripts and running tasks much easier and faster. It uses advanced AI technology and follows a simple "first-come, first-serve" system for handling files. Developers and researchers can get more work done because this tool takes care of routine jobs automatically. By handling the repetitive tasks, it lets technical professionals focus on more important work instead. We named this felix.

CHAPTER 2
OBJECTIVES

Real time operating system:

An LLM OS has the capability to enhance task prioritization, resource allocation, and adaptive system tuning using natural language commands. Users can instruct the OS to optimize CPU usage for time-sensitive tasks, monitor real-time performance, and debug system behavior. The LLM can also predict workload patterns and dynamically adjust system parameters to reduce latency and power consumption.

Process Scheduling Simulator:

Process Scheduling Simulator in an LLM OS can allow users to switch between scheduling algorithms like FCFS, SJF, or Round Robin. This shell can analyze real-time CPU load, optimize process distribution, and suggest the best scheduling strategies for different workloads. It can also provide insights into scheduling efficiency, explaining how changes affect system performance.

Key features:

Conversion of natural language into a python code: The shell takes a natural language input from the user and converts it into a python code, so that the shell can understand what its task is.

Automatic dependency extraction and installation: The shell next installs all the required pip libraries that are needed to run the overall LLM.

File management using Ready, Running, and Finished folders: We store the files to simulate the process scheduling. In this context, the files are analogous to the processes. Once a prompt is given, the shell then stores all the required files in the respective folders, to show that either it is in ready state, running state or finished state.

FCFS scheduling for task automation: We use the FCFS scheduling algorithm to schedule the files needed to run a particular prompt.

Interactive terminal interface with system commands support: We use a simple interactive terminal to communicate with felix.

CHAPTER 3

METHODOLOGY AND SYSTEM DESIGN BLOCK DIAGRAM

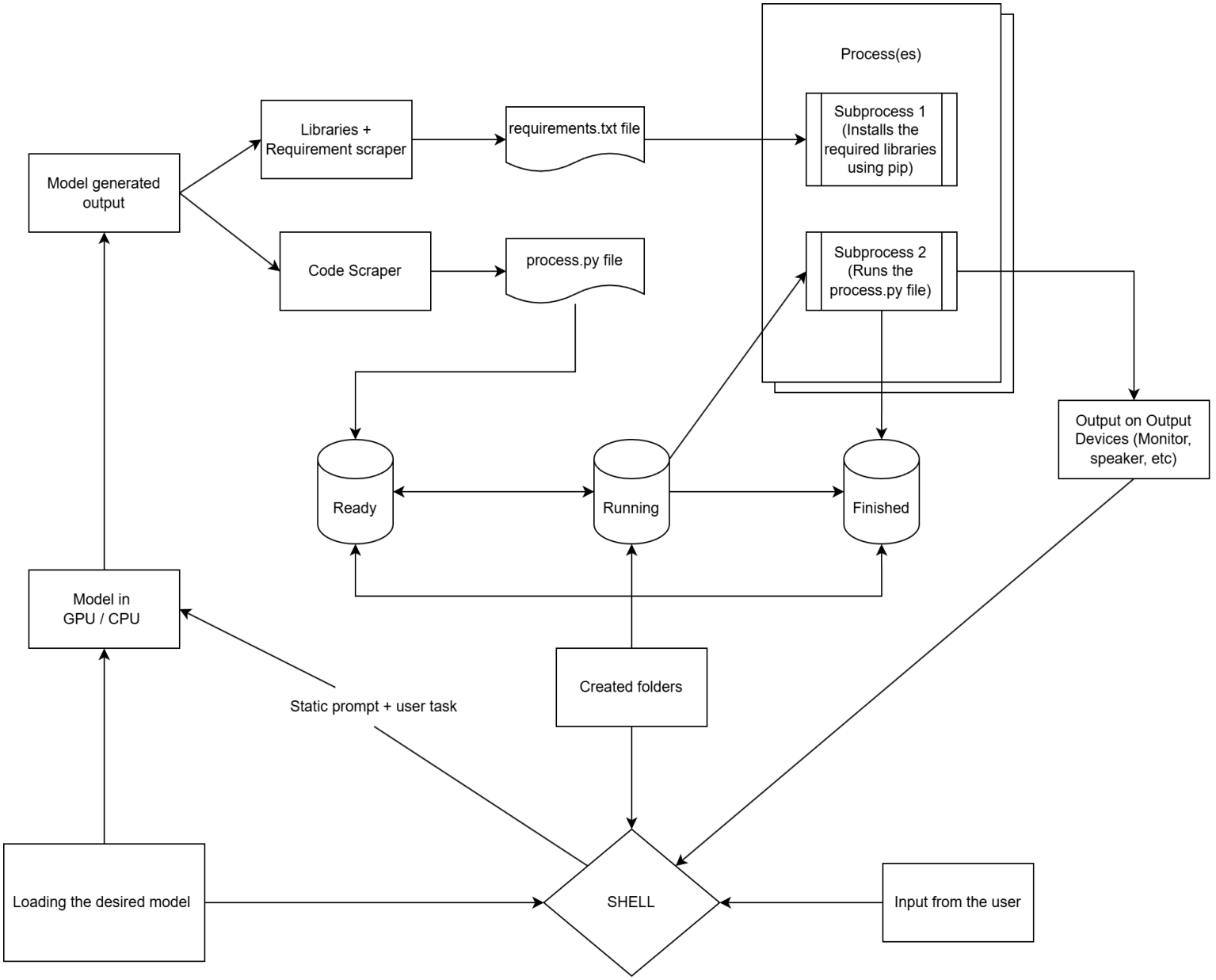

Fig [1]: Methodology Diagram

Methodology for this system begins at the foundational level with user interaction through a shell interface, where users can input commands and tasks. Simultaneously, the system loads the desired AI model into available GPU or CPU memory, preparing the

computational foundation needed for processing complex language instructions and generating appropriate outputs.

Once activated, the AI model generates output that follows a bifurcated processing pathway designed for maximum efficiency. The first path directs output through a Libraries and Requirement scraper that meticulously analyzes the model's output to identify necessary dependencies, culminating in the creation of a structured requirements.txt file. Parallel to this, a second pathway channels the model's output through a Code Scraper component that extracts and formats executable code, producing a functional process.py file that contains the programming logic needed to fulfill the user's request.

The methodology incorporates a robust file status management system consisting of three distinct database states: Ready, Running, and Finished. This tripartite structure enables precise tracking of each process throughout its lifecycle, ensuring proper resource allocation and preventing execution conflicts. The folders communicate with each other to maintain synchronization and facilitate smooth transitions between process states as execution progresses.

The execution layer of this methodology employs two subprocesses working in concert to deliver results. Subprocess 1 takes responsibility for dependency management, automatically installing all required libraries identified in the requirements.txt file using Python's pip package manager. Following successful installation, Subprocess 2 activates to execute the previously generated process.py file, applying the created code to perform the requested task without requiring manual intervention from the user.

To maintain organization and facilitate future reference, the system implements a structured folder creation mechanism that generates and populates directories with relevant files and artifacts produced during execution. This organizational approach ensures that outputs and intermediate results remain accessible and logically arranged for subsequent use or modification.

The final stage of this methodology involves comprehensive output handling, where the results of execution are formatted and channeled to appropriate output devices such as

monitors or speakers. This creates a complete operational cycle where user input triggers processing, which generates executable components that are automatically installed and run, with results seamlessly presented back to the user through the shell interface, thus completing the interactive loop of this AI shell system.

CHAPTER 4

IMPLEMENTATION AND RESULTS

The Python script implements an advanced AI-powered shell system that serves as an interface between natural language commands and executable Python code. At its core, the implementation leverages a DeepSeek language model (specifically deepseek-coder-1.3b-instruct) to translate user prompts into functional Python scripts. The code establishes a file management framework with three distinct directories ("ready_files," "running_files," and "finished_files") that systematically track the execution status of generated scripts throughout their lifecycle.

The system employs a First-Come-First-Served (FCFS) scheduling algorithm implemented through a deque data structure from the collections module, ensuring fair and sequential execution of all generated scripts. Upon initialization, the program displays an elaborate boot sequence with stylized loading messages that simulate the activation of various AI components, creating an immersive terminal experience. The loading animation runs in a separate thread to allow simultaneous model loading, demonstrating thoughtful use of threading for improved user experience.

A particularly innovative aspect of this implementation is its automatic dependency management system, which uses regular expressions to scan generated code for import statements. It then filters out standard library modules and creates a requirements.txt file for any external dependencies. This file is subsequently used with pip to automatically install necessary packages before script execution, eliminating the need for manual dependency management.

The code generation process begins when a user prefixes their natural language prompt with "felix," triggering the system to formulate a structured prompt for the DeepSeek model. The model's response is processed to extract clean executable code, which is then saved to a timestamped Python file in the ready directory. Code extraction employs regex pattern matching to isolate code blocks from any surrounding explanatory text that might be included in the model's response.

When executing scripts, the system maintains precise state tracking by moving files between directories to indicate their current status. The shell also preserves traditional command-line functionality, passing through standard shell commands to the underlying operating system while handling special commands like "cd" directly. The implementation includes robust error handling for keyboard interrupts and EOF conditions to ensure graceful termination of the program when requested.

The GPU acceleration capability is particularly noteworthy, as the script automatically detects CUDA availability and optimizes model loading accordingly. When a GPU is present, the model leverages torch.bfloat16 precision to balance performance and memory usage. The system gracefully falls back to CPU execution when necessary, displaying appropriate warnings to the user. This adaptive resource management demonstrates consideration for varied hardware environments.

The user interface is enhanced with thoughtful details such as a typewriter effect for text output, color-coded status messages, and emoji indicators that provide visual feedback on operation success. These elements combine to create an engaging and intuitive interaction model that balances technical functionality with user-friendly design. Through its integration of advanced AI capabilities, efficient resource management, and intuitive user experience considerations, this code presents a sophisticated approach to AI-assisted script generation and execution within a terminal environment.

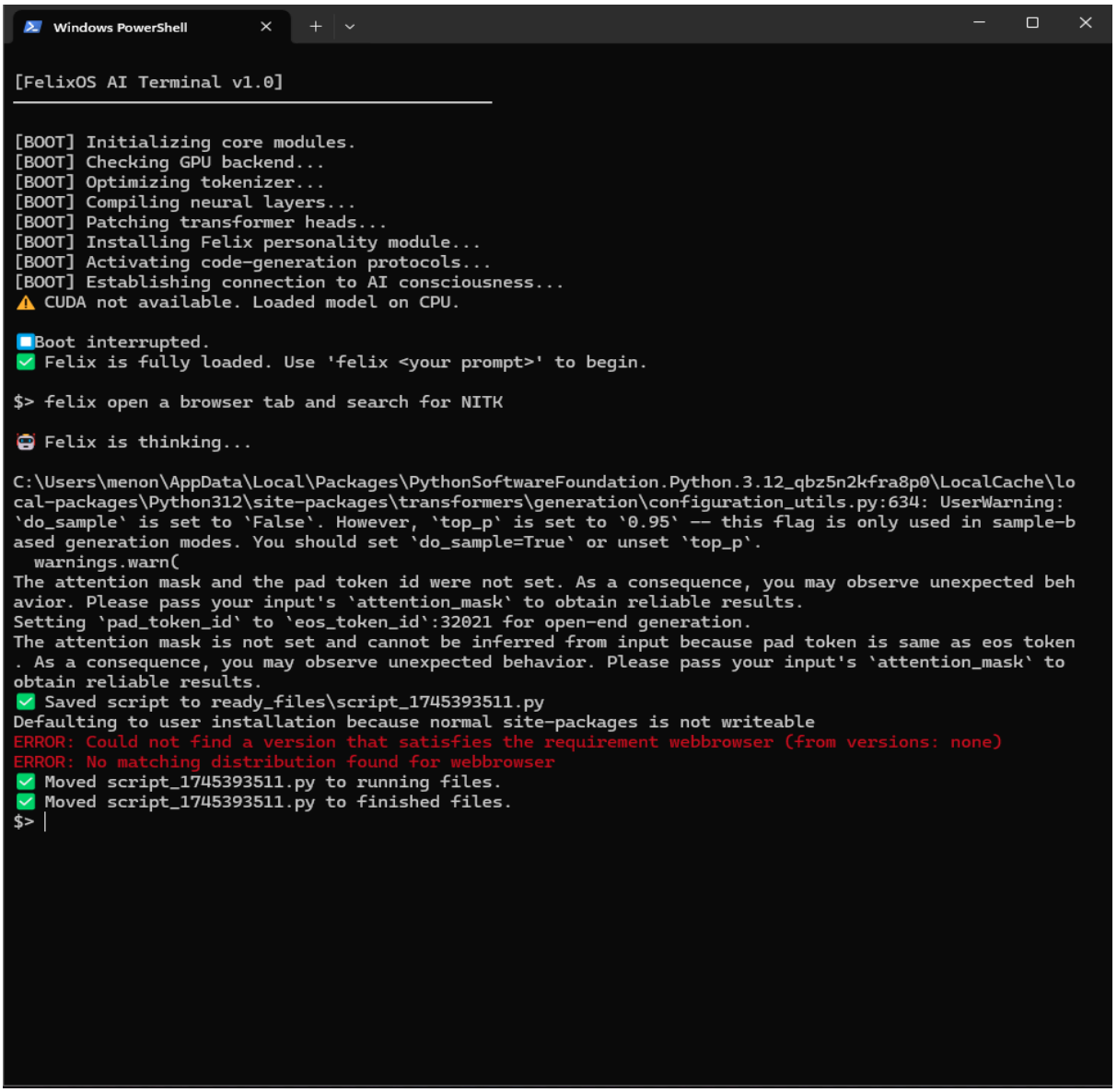

Fig [2.1]: Shell Output

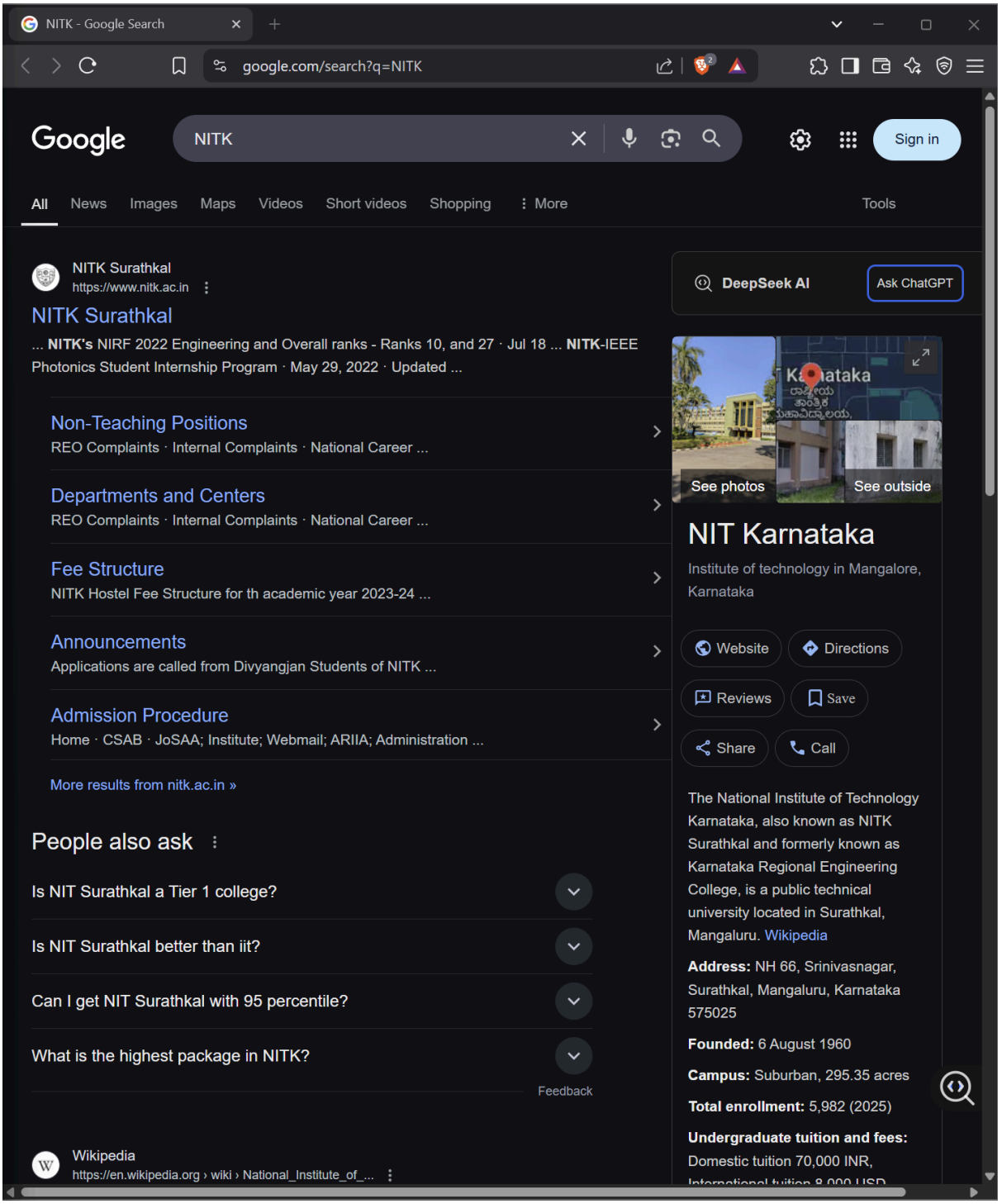

Fig [2.2]: Browser Output

CHAPTER 5
CONCLUSION AND FUTURE WORK

The current implementation of the AI shell system provides a solid foundation for code generation and execution, but several avenues for enhancement could significantly expand its capabilities and user experience. The proposed improvements in scheduling algorithms, error handling, GUI integration, and multi-language support represent important first steps in this evolution.

Implementing additional scheduling algorithms beyond the current First-Come-First-Served approach would enhance the system's flexibility in handling diverse workloads. Priority-based scheduling could allow urgent tasks to receive immediate attention, while Round Robin scheduling would ensure fair distribution of computational resources among multiple scripts. These alternatives would make the system more adaptable to complex workflow requirements and user priorities.

Enhanced error handling and logging capabilities would significantly improve both the robustness and maintainability of the system. By implementing comprehensive error capture mechanisms with detailed logging, the system could provide more informative feedback to users when scripts fail and maintain audit trails for debugging. This would transform the AI shell from a simple code generation tool into a reliable production environment.

GUI integration for non-terminal users represents a critical accessibility improvement that would broaden the system's appeal beyond command-line enthusiasts. A graphical interface could provide intuitive access to the AI shell's capabilities, visualization of execution status, and easier review of generated code, making the technology accessible to developers who prefer visual interfaces over terminal environments.

Support for multi-language code generation would dramatically expand the utility of the AI shell across diverse development ecosystems. By extending beyond Python to languages such as JavaScript, Java, C++, and Rust, the system could serve a broader range of development needs and become a universal coding assistant regardless of a project's technical stack.

Integration of version control capabilities would transform the AI shell into a more comprehensive development environment. By implementing Git functionality directly within the system, users could track changes to generated code, create branches for experimental features, and seamlessly commit successful implementations. This would not only preserve the history of AI-generated solutions but also facilitate collaboration among multiple users working with the same AI shell instance.

Performance optimization through caching mechanisms represents another promising development direction. By storing previously generated code snippets and their corresponding prompts, the system could build a local knowledge base that reduces reliance on the language model for repetitive tasks. This approach would significantly decrease response time for common queries while reducing computational overhead and potential costs associated with continuous model inference.

The AI shell system advances natural language processing integration with software development by automating Python code generation from prompts, streamlining development for users of all skill levels. Its robust file management, dependency handling, and execution pipeline demonstrate its viability, while future enhancements like scheduling algorithms, error handling, GUI integration, and multi-language support could expand it into a comprehensive development tool. As AI evolves, such systems will likely become essential, bridging natural language and technical implementation, and maturing into versatile assistants for the entire software lifecycle from conception to execution and iterative improvement.