International Journal of Scientific Research and Engineering Development-– Volume 9 Issue 1, Jan-Feb 2026

Available at www.ijsred.com

RESEARCH ARTICLE OPEN ACCESS

# AI-Driven Operating System for Intelligent Desktop Automation

M. Vishnu Vardhan[1], Mohammed Junaid[2], Mounish[3], T. Vishnu Vardhan[4], Mr.Iliyaz Pasha M[5]

[5]Assistant Professor, Department of Computer Science and Engineering,
R.L. Jalappa Institute of Technology, Doddaballapur, Karnataka, India
Student, Department of Computer Science and Engineering

----------**********----------

## Abstract:

Modern desktop operating systems require significant command-line knowledge and manual configuration, creating accessibility barriers for many users. This paper presents an AI-driven operating system assistant prototype that integrates a locally-deployed Large Language Model into a minimal Linux environment to enable natural language control of system operations. The proposed architecture employs a modular design with a user interface, a central Orchestrator for secure command execution, and a fine-tuned LLM that translates user intent into structured system actions. All processing occurs offline on consumer hardware, preserving privacy while maintaining responsiveness. The prototype was evaluated through desktop automation scenarios including file management, project organization, and version control workflows, demonstrating the feasibility of using compact, domain-adapted language models for intuitive desktop interaction. The results establish a foundation for future research into intelligent, offline-capable computing assistants.

**Keywords—** AI-Driven Operating System, Large Language Model (LLM), Natural Language Interface, Offline AI Desktop Assistant, System Automation

----------**********----------

## I. INTRODUCTION

Traditional desktop operating systems rely heavily on command-driven interaction, manual configuration, and user expertise to perform even routine system tasks. Users must remember complex terminal commands, navigate filesystem hierarchies, install and manage software manually, and troubleshoot unexpected errors through log inspection. While this workflow is flexible for power users, it presents a steep learning curve for the majority of everyday computer users who primarily seek a seamless, intuitive, and supportive computing experience.

Recent advances in Large Language Models (LLMs) have demonstrated strong capabilities in natural-language understanding, intent interpretation, and procedural reasoning. These developments open the opportunity to fundamentally rethink how users interact with their personal computers. Instead of acting as passive platforms, operating systems can evolve into active participants that understand user goals, automate workflows, and assist in troubleshooting—without requiring technical expertise.

  

This paper presents a prototype for an AI-assisted desktop automation system centred around an offline, on-device intelligent assistant capable of translating natural language instructions into actionable, validated system operations. The system is built around a modular architecture involving a User Interface, an Orchestrator, and a fine-tuned, lightweight LLM, forming a closed, privacy-preserving workflow that functions without internet access. The Orchestrator ensures secure execution through structured action parsing, strict privilege isolation, and a controlled terminal environment.

The prototype demonstrates a personal-computer environment that is intuitive, autonomous, and helpful. It enables users to manage files, organize projects, automate repetitive tasks, and interact with their system through natural language. For example, a user working on a college project can request the assistant to "organize my project folder, initialize a Git repository, and prepare it for upload to GitHub," demonstrating how the system can execute these steps safely and transparently. By integrating AI into the core interaction loop, the proposed system aims to enhance usability, reduce cognitive workload, and make advanced system operations more accessible. This research explores how combining lightweight AI models with a secure execution framework can contribute to redefining the user experience of modern personal computing environments.

## II. LITERATURE SURVERY

[1] Vaswani et al: Vaswani et al. introduced the Transformer architecture, demonstrating that self-attention mechanisms can replace recurrence and convolution for sequence modeling. Their work showed how multi-head attention and positional encoding enable efficient modeling of long-range dependencies and contextual relationships. This architecture forms the foundational design of modern large language models, and our system adopts transformer-based language modeling to enable contextual understanding of multi-step operating system instructions and conversational continuity.

[2] Devlin et al: Devlin et al. introduced BERT, a bidirectional transformer-based language representation model that significantly advanced natural language understanding tasks. Their work demonstrated how deep contextual embeddings enable models to capture semantic relationships and syntactic structure more effectively than traditional sequential models. This research established the foundation for intent understanding and contextual interpretation, which is essential for accurately interpreting natural-language system commands in AI-driven operating system environments.

[3] Brown et al: Brown et al. demonstrated that large-scale language models exhibit few-shot learning capabilities through prompt-based conditioning, without requiring task-specific fine-tuning. Their work showed that a single model can generalize across diverse tasks using only natural language prompts. This insight directly influences our prompt-driven interaction design, allowing the assistant to perform a wide range of operating system tasks using a unified model without retraining for each function.

[4] Touvron et al: Touvron et al. introduced LLaMA, a family of open and computationally efficient foundation language models designed for local deployment. Their work emphasized parameter efficiency and high performance on consumer-grade hardware. This research informs our decision to use locally executable language models, enabling offline operation, reduced resource overhead, and privacy-preserving AI-assisted system interaction.

  

[5] Yao et al: Yao et al. proposed the ReAct framework, which synergizes reasoning and action execution by interleaving intermediate reasoning steps with tool usage. Their approach demonstrated improved task performance by enabling models to plan and adapt based on observations. Our system draws conceptual inspiration from ReAct by separating reasoning and action phases, while introducing a dedicated orchestrator to validate and safely execute system commands generated by the model.

[6] Wei et al: Wei et al. introduced Chain-of-Thought prompting, showing that explicitly generating intermediate reasoning steps significantly improves reasoning accuracy in large language models. Their work highlights the importance of structured reasoning in complex decision-making tasks. In our system, Chain-of-Thought reasoning is used internally to improve planning and command generation reliability, while remaining hidden from the user interface to maintain a clean and minimal interaction experience.

[7] Schwartz-Narbonne et al: Schwartz-Narbonne et al. investigated formal methods for the safe execution of learned control policies, emphasizing correctness, verification, and execution constraints in autonomous systems. Their work underscores the importance of enforcing safety boundaries when integrating learned behaviors into real-world systems. These principles directly influence our execution model, where all model-generated actions are mediated through an orchestrator that enforces isolation, privilege control, and structured execution policies.

[8] Gulzar et al: Gulzar et al. explored automated detection and correction of configuration errors in Linux systems, demonstrating how intelligent analysis of system state can reduce operational faults. Their findings motivate the diagnostic capabilities of our assistant, which analyzes configuration outputs and system state information before suggesting corrective actions, rather than applying direct or potentially unsafe modifications.

[9] Mickens et al: Mickens et al. introduced Pivot, a system for fast and isolated execution of code fragments using generator chains. Their work emphasized isolation and controlled execution as core principles for secure system integration. This research informs our modular system design, where the user interface, reasoning engine, and command execution environment are isolated components to prevent fault propagation and enhance system stability.

[10] Myers et al: Myers et al. examined natural programming languages and environments that enable users to express computational intent using human-centric language rather than rigid syntax. Their work argued for systems that adapt to human communication patterns. This perspective directly motivates our natural language–first operating system interface, allowing users to interact with system functionality conversationally instead of relying on traditional command-line tools.

## III. MOTIVATION AND REAL WORLD SCENARIO

Modern personal computers remain powerful but often unintuitive tools: they execute commands exactly as instructed, with limited contextual understanding of user intent, and depend heavily on the user's technical expertise to perform moderately complex tasks. Whether managing files, organizing projects, or configuring software environments, users frequently encounter repetitive, time-consuming workflows that disrupt productivity.

  

A typical example occurs during local software development. A student or developer may need to initialize a Git repository, create a structured project folder, and prepare it for version control—a process that requires remembering numerous CLI commands, interpreting error logs, and manually navigating between tools. Even minor mistakes, such as incorrect paths or missing permissions, can halt progress and require extensive troubleshooting.

To address this friction, we propose an AI-assisted desktop automation system that embeds an offline, intelligent assistant capable of understanding and executing natural language instructions. Instead of requiring users to memorize commands or interpret cryptic errors, the system enables conversational interaction—for example, a user could request: "Create a new C++ project folder with src, docs, and scripts directories, and initialize a Git repository." The system interprets the request, translates it into validated system actions, executes them through a secure Orchestrator, and returns the results. By enabling offline AI reasoning and controlled automation of common desktop tasks, this approach aims to reduce the technical burden on users, enhance accessibility for non-experts, and improve productivity for students, developers, and everyday computer users.

## IV. CONTRIBUTION

This work demonstrates a prototype system for enabling natural-language interaction with desktop operating systems through a secure, locally-hosted AI assistant. The primary contributions are:

An architecture for secure, offline AI-assisted desktop automation, centred on a dedicated Orchestrator layer that mediates between a local LLM and the OS, ensuring command validation and execution safety without cloud dependence.

- **A structured natural-language-to-command translation approach**, in which user intents are mapped to JSON-formatted executable actions, enabling deterministic parsing and controlled execution.
- **Implementation and evaluation of a fully local automation pipeline**, showing that lightweight, fine-tuned LLMs can perform common desktop tasks—such as file management, project setup, and version control operations—entirely offline.
- **A modular prototype showcasing clear separation of concerns** between the user interface, reasoning model, and execution sandbox, illustrating a replicable design for privacy-preserving desktop AI assistants.
- **Evidence that compact models can support adaptive command execution** through a feedback loop where terminal output is analyzed to refine subsequent actions, improving reliability in multi-step workflows.
- **A publicly documented methodology** for dataset creation, model fine-tuning, and system integration that can serve as a reference for future work in offline, language-driven OS interaction.

## V. SYSTEM ARCHITECTURE

### A. User Interface Layer:

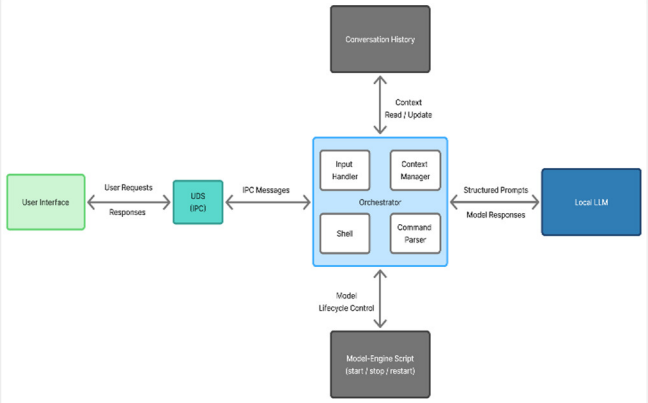


*Fig-1 High-level architecture of implementation illustrating the interaction between the User Interface, Orchestrator, local LLM engine, and the command execution environment.*

The User Interface (UI) serves as the primary point of interaction, designed as a lightweight, chat-style

  

desktop assistant that operates entirely offline. Users enter natural-language requests—such as file organization, project setup, or basic automation tasks—into a simple text prompt. The UI transmits all input directly to the Orchestrator for processing and displays structured responses, explanations, and task progress. For transparency and debugging, an optional terminal-mirror mode allows users to view command execution in real time. The interface is intentionally minimal to reduce system overhead and maintain responsiveness on consumer hardware. Crucially, the UI does not execute any commands; it acts solely as a safe interaction surface, with all decision-making and execution handled by subsequent layers.

### B. Orchestrator Layer (Central Control Unit)

The Orchestrator functions as the core control mechanism, connecting the UI, the LLM, and the underlying Linux environment. It ensures that every operation follows a structured, secure workflow. The Orchestrator operates in two modes: Normal Mode for efficient execution, and Debug Mode, which logs the entire interaction pipeline for analysis and troubleshooting.

The Orchestrator consists of several key modules:

- **Request Handling Module**

Receives user input from the UI, normalizes it, and attaches relevant metadata (such as conversation state and previous command output) to create a context-rich prompt for the LLM.

- **Communication Bridge (Model-Engine):** Manages the lifecycle of the local LLM server, handling startup, shutdown, resource checks, and health monitoring to ensure low-latency availability.
- **Safety & Validation Module:** Performs syntax validation on model-generated commands and enforces privilege isolation by executing all commands in a controlled environment. The current implementation focuses on basic validation, while the modular design allows for future integration of more advanced security measures—such as container-based sandboxing, system-call filtering, or path-restriction policies—as needed.
- **Terminal Execution Environment:** Uses an embedded tmux session to run validated commands in an isolated, non-root context. This provides sandboxed execution while capturing complete terminal output (stdout and stderr) for feedback and logging.
- **Logging & Audit Subsystem:** Records all inputs, outputs, commands, model responses, and validation decisions, supporting traceability, debugging, and safety auditing—especially when Debug Mode is active.
- **Extensibility Framework:** The Orchestrator's architecture is designed to support pluggable modules for future enhancements, such as distributed communication protocols that could enable coordination between multiple instances of the system across a local network or edge devices.

Through these components, the Orchestrator transforms natural-language input into validated, secure system operations, forming the foundation of the prototype's intelligent behaviour.

### C. Model Layer (Local LLM Engine)

The Large Language Model (LLM) serves as the core reasoning component of the prototype. We use a lightweight, fine-tuned variant (1B–2B parameters) designed for offline execution on consumer hardware, enabling low-latency inference while preserving user privacy and eliminating cloud dependency. The model receives pre-processed user intent and contextual metadata from the Orchestrator and generates responses according to a strictly defined, three-part format that separates reasoning, user communication, and executable instructions. This structured approach maintains

  

safety, transparency, and predictability throughout the automation pipeline.

- **Structured Model Response Format**

Each model response is organized into three distinct sections:

[THINKING]: Contains the model's internal reasoning and step-by-step planning. This section is never executed and serves only to improve multi-step reasoning quality and decision consistency.

[USER]: Provides natural-language explanations, summaries, or confirmations intended for display in the user interface.

[ACTIONS]: Contains strictly structured, machine-readable instructions in JSON format. Only this section is consumed by the Orchestrator, where each proposed action undergoes validation before execution in the controlled terminal environment.

```
[THINKING]
<internal reasoning and planning based on user intent and system state>
[USER]
<human-readable explanation or confirmation message shown to the user>
[ACTIONS]
[
{"action": "execute_command", "command": "<shell command>"},
{"action": "send_keys", "keys": "<interactive input>"}
]
```

Fig-1. Illustrates the end to end execution pipeline, showing LLM reasoning, structured action generation, sandboxed command execution inside tmux, terminal output analysis, and the final user facing response.

- **Rationale for the Sectioned Response Format**

Instead of emitting a single monolithic JSON object, the sectioned response format is employed to address several practical limitations inherent in OS-level automation. First, it enforces a clear separation of responsibilities by isolating reasoning, explanations, and executable commands into distinct blocks, which reduces parsing complexity and prevents the accidental execution of non-operational content. Second, by extracting only the [ACTIONS] block, the Orchestrator can apply consistent validation rules, thereby improving safety and determinism while minimizing the risk of unintended command execution caused by malformed or ambiguous model output. Third, the format remains parsable even when individual sections contain minor errors, granting the system robustness against partial or invalid output and allowing it to recover gracefully or request correction without crashing the pipeline. Fourth, explicit reasoning traces in the [THINKING] section enable transparent debugging and auditing, letting developers inspect why a particular command was proposed without exposing internal logic to the execution layer. Finally, this clean separation facilitates a closed-loop feedback mechanism: terminal output can be returned to the model, which can then revise specific sections while preserving the overall response structure, thereby supporting iterative self-correction during multi-step workflows.

- **Model Training and Adaptation**

The LLM is fine-tuned on a domain-specific dataset of approximately 4,000 samples covering system-administration dialogues, command-generation tasks, and workflow-automation examples. This training enables the model to interpret user intents related to file management, project organization, version-control operations, and basic system navigation. When command execution produces

  

errors or unexpected output, the Orchestrator returns structured terminal feedback to the LLM, which can then generate a revised response following the same three-part format. This closed-loop interaction demonstrates the potential for adaptive, self-correcting behavior in desktop-automation scenarios.

Overall, the Model Layer provides the intelligent reasoning foundation for natural-language interaction with desktop systems. By enforcing a strict response contract and delegating all execution authority to the Orchestrator, the prototype achieves a balance between autonomy, safety, and transparency in offline desktop automation.

## VI.METHODOLOGY

Our prototype was developed through four coordinated phases: dataset creation, model fine-tuning, orchestrator implementation, and system integration. Each phase was guided by the requirements of offline operation, basic safety, and compatibility with consumer hardware.

### A. Dataset Creation

A domain-specific dataset of approximately 4,000 multi-turn dialogues was constructed to train the LLM for interactive desktop automation. Each dialogue simulates a realistic user-assistant exchange and is structured to reinforce the three-part response format ([THINKING], [USER], [ACTIONS]). The data includes natural-language queries, model reasoning traces, user-facing explanations, and executable command sequences, along with simulated terminal output to enable feedback-driven adaptation. For instance:

```
{
"dialogue": [
 {
"user_input": "<natural language request>",
  "thinking_output": "<model's internal
reasoning>",
  "user_output": "<explanation for user>",
  "orchestrator_output": "[{
        \"action\": \"execute_command\",
        \"command\": \"<shell command>\"}]"
 },
 {
  "orchestrator_input":
"[TERMINAL_OUTPUT]\n<simulated terminal
output>",
  "thinking_output": "<analysis of terminal
output>",
  "user_output": "<summary for user>",
  "orchestrator_output": null
 }
 ]
}
```

### B. Model Fine-Tuning

A compact LLaMA-based model (1B–2B parameters) was selected for its suitability for local execution on laptops with limited resources. Fine-tuning was performed using LoRA (Low-Rank Adaptation) on a single T4 GPU (Google Colab), with a custom training format that mirrors the dialogue structure shown above. The objective was to teach the odel to: (1) interpret natural-language intents for common desktop tasks, (2) produce structured three-part responses ([THINKING], [USER], [ACTIONS]), and (3) adapt its reasoning based on terminal feedback. Training leveraged the simulated terminal output included in the dataset to enable basic error-recovery behavior. After fine-tuning, the model demonstrated improved accuracy in command generation and the ability to refine its outputs when presented with execution results.

### C. Orchestrator Development

The Orchestrator, implemented in Go, serves as the central control unit that mediates between the user interface, the LLM, and the underlying OS. Its modular design includes:

  

- **Request Handler:** Formats user input into context-rich prompts for the LLM**.**
- **Safety & Validation Module**: Performs basic syntax checks on model-generated commands to ensure they are appropriate for execution.
- **Terminal Execution Environment:** Manages a persistent, isolated tmux session that provides sandboxed command execution and complete output capture, preventing direct LLM access to the OS.
- **Logging and Debugging Subsystem:** Supports two operational modes; Normal (minimal overhead) and Debug (detailed traces of user prompts, model responses, validated actions, and terminal output)—enabling analysis without impacting runtime performance.
- **Model-Engine Module:** A dedicated helper script that manages the local LLM server lifecycle, monitoring health and ensuring low-latency communication.

This architecture ensures that all commands are validated and executed within a controlled environment while keeping all processing offline.

## VII. IMPLEMENTATION

The proposed AI-driven operating system was implemented using a modular and incremental development approach, integrating a locally deployed Large Language Model (LLM), a centralized Orchestrator framework, and a hardened base operating system into a unified intelligent desktop environment. Each subsystem was developed, validated, and refined independently prior to full-stack integration, ensuring stable operation, basic safety guarantees, and complete offline functionality.

### A. Local LLM Deployment and Execution Control.

The system utilizes a fine-tuned 1-billion-parameter LLaMA-based language model deployed locally using the llama.cpp inference framework. The model is exposed as an HTTP-based inference service through the native llama.cpp serving interface, enabling CPU-optimized execution on consumer-grade hardware without reliance on cloud services or GPU acceleration.

A lightweight model engine control script manages the complete lifecycle of the model server, including startup, shutdown, restart, health monitoring, and model selection. The Orchestrator continuously monitors the availability of the model server and initiates it on demand when user requests are received, thereby optimizing system resource utilization during idle periods.

Communication between the Orchestrator and the language model occurs exclusively over a local HTTP interface. Model responses are constrained to a structured three-part format comprising [THINKING], [USER], and [ACTIONS] sections. The [THINKING] component contains internal reasoning annotations used during development and debugging, the [USER] section provides human-readable explanations intended for user feedback, and the [ACTIONS] section contains structured, machine-readable commands encoded in JSON format. Only the [ACTIONS] section is forwarded to the execution pipeline, ensuring deterministic parsing, reliable validation, and strict separation between reasoning and execution.

  

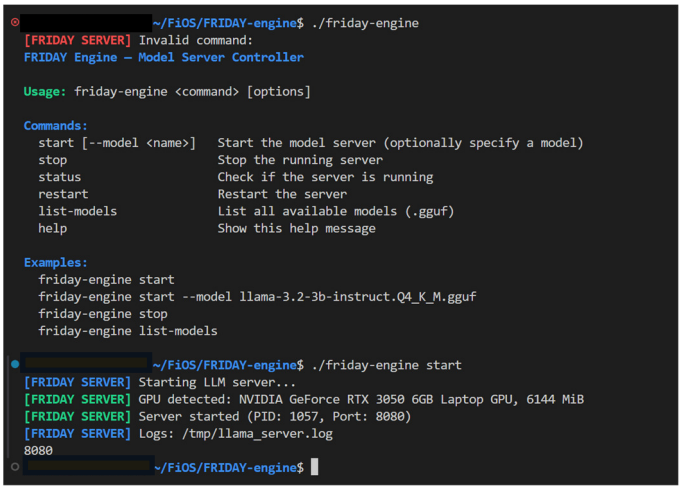


*Fig-2 Local LLM serving architecture using llama.cpp and HTTP-based inference interface*

## B. Orchestrator Architecture and Control Flow.

The Orchestrator, implemented in Go to leverage its low runtime overhead and native concurrency support, functions as the central control layer of the system. It mediates all interactions between user interfaces, the local language model server, and the underlying operating system, providing a structured execution flow and centralized control over command dispatch and logging.

User requests are received through a Unix Domain Socket interface, enabling low-latency local inter-process communication. The Orchestrator supports concurrent client connections while maintaining thread-safe internal state. Structured prompt contexts are constructed by combining system prompts, a bounded interaction history, and the current user query before being forwarded to the language model server. A lightweight state-tracking mechanism, comprising idle, processing, waiting, error, and completion states, is used to monitor request progression and provide consistent status updates to connected interfaces.

In the current prototype, model-generated actions are parsed and inspected prior to execution to ensure structural correctness and adherence to the expected action format. Commands are executed within a non-root user context to limit system exposure. More advanced validation mechanisms—such as semantic command analysis, detection of destructive patterns, privilege-escalation prevention, and policy-based execution control—are intentionally deferred and identified as directions for future work.

Command execution is performed within a dedicated and persistent *tmux* session managed by the Orchestrator, allowing isolation from the user's interactive shell while enabling full capture of command output. Execution completion is inferred through shell prompt detection, providing a practical, non-intrusive mechanism for determining command termination. Standard output and error streams are captured and returned to the Orchestrator, where they are either presented to the user or supplied back to the language model to support iterative reasoning and corrective response generation.

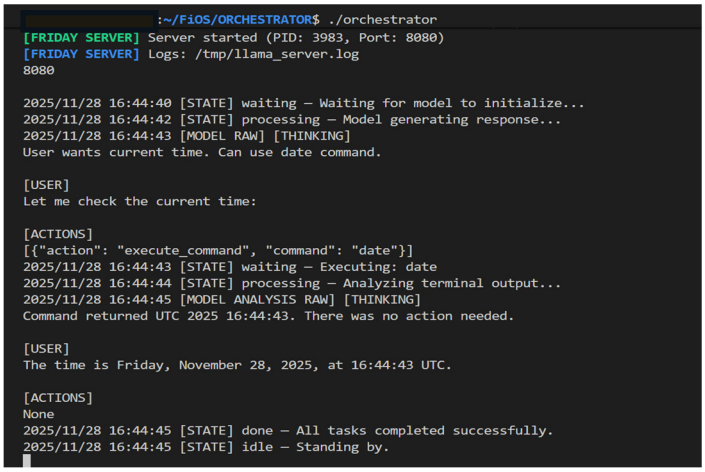


*Fig-3 Orchestrator-controlled tmux execution pipeline and feedback loop*

The Orchestrator supports two operational modes to balance performance and observability. **Normal**

  

**mode** enables minimal logging to reduce runtime overhead, while **Debug mode** records detailed traces including model prompts, parsed responses, validation decisions, executed commands, and terminal output. This dual-mode design enables effective debugging and system analysis without requiring modifications to the core architecture.

### C. User Interaction Layer.

Two complementary user interface implementations were developed to evaluate system usability and interaction flexibility.

A lightweight command-line interface supports rapid testing and real-time inspection of system behaviour. The interface streams structured responses from the Orchestrator and clearly distinguishes between reasoning output, proposed actions, execution status, and terminal results.

A simple graphical desktop assistant interface provides a natural-language input field, a scrollable interaction history, and clear visualization of executed commands and their outcomes. The interface operates entirely offline and communicates with the Orchestrator using the same local socket-based protocol.

### D. Base Operating System Preparation.

A minimal Debian-based environment was prepared as the target deployment platform, referred to as the Golden Master base operating system. Non-essential packages were removed to reduce the system attack surface, while essential system utilities, tmux, and LLM runtime dependencies were retained. User permissions were configured to enforce non-root execution and controlled system access. The Golden Master OS is currently deployed within a controlled virtual machine for stability validation, with bare-metal deployment planned for future phases.

### E. System Integration and Validation.

Following full system integration, an iterative testing phase was conducted to validate end-to-end functionality. Comprehensive tests verified the complete LLM–Orchestrator–execution pipeline, while stress tests evaluated latency, concurrency handling, and overall system responsiveness. Additional safety evaluations confirmed consistent generation of warnings for potentially destructive commands and strict enforcement of execution constraints by the Orchestrator prior to command execution.

## VIII. RESULTS AND DISCUSSION

This section presents observations derived from controlled execution traces collected during interaction with the prototype system. The evaluation focuses on **end-to-end task latency and execution behaviour** as experienced by the user, rather than establishing absolute performance guarantees. The reported results serve as **empirical reference measurements**, intended to demonstrate system feasibility and architectural behaviour under realistic usage conditions.

All measurements were obtained from real executions of the implemented system and reflect the combined effects of model inference, command validation, execution, and output handling.

### A. Experimental Setup and Benchmarking Scope

The benchmarking experiments were conducted on a consumer-grade system equipped with an Intel Core i5-13450HX processor, 16 GB of RAM, and an NVIDIA RTX 3050 GPU with 6 GB VRAM. Although a discrete GPU was available, the system primarily relied on CPU-based inference using the llama.cpp backend, reflecting a practical offline deployment scenario.

It is important to emphasize that absolute latency values are highly dependent on system configuration,

  

including processor architecture, available memory, storage performance, background load, and model size. Consequently, the measurements reported in this section should not be interpreted as fixed performance benchmarks. Instead, they are provided as observational reference values, demonstrating how the system behaves on one representative hardware configuration.

The benchmark dataset consists of multiple natural-language requests grouped into four functional categories:

- file operations,
- system information queries,
- development-related tasks, and
- package and tool operations.

For each task, a single end-to-end latency value was recorded, measured from the point at which the Orchestrator began processing the request to the completion of command execution and response generation.

## B. End-to-End Latency Across Task Categories

Figure **4** presents the **average end-to-end latency observed for each task category**.

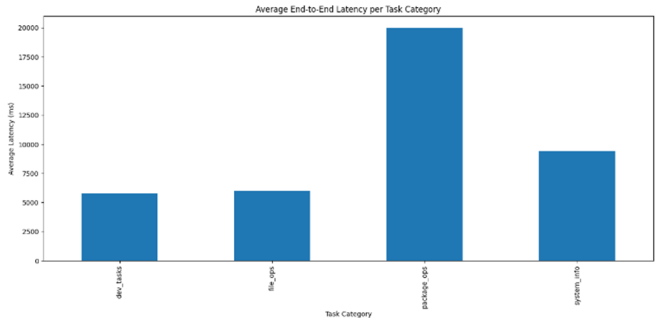


Fig-4 Average End-to-End Latency per Task Category

File operations and development-related tasks generally exhibit lower average latency, as these requests typically translate into lightweight shell commands with limited output. System information queries demonstrate moderately higher latency due to increased command execution and output processing.

Package and tool operations show the highest average latency. These tasks often involve querying package managers, enumerating system services, or accessing documentation, all of which produce larger outputs and incur additional I/O and parsing overhead. The observed increase in latency is therefore attributable to command complexity rather than instability within the system architecture.

## C. Latency Variability and Distribution

Figure **5** illustrates the **distribution of task latency across categories** using box plots.

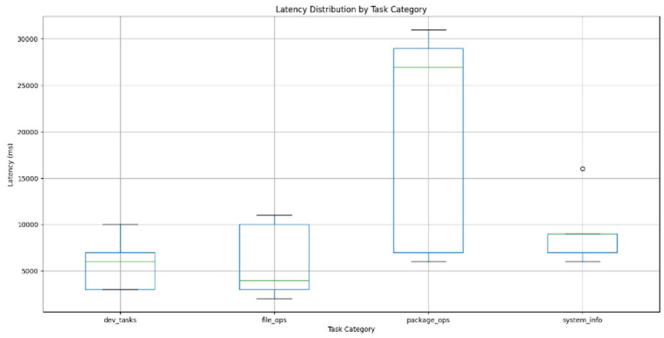


*Fig-5 Latency Distribution by Task Category*

File and development tasks exhibit relatively tight latency distributions, indicating predictable execution behaviour. In contrast, package-related operations show greater variability, including higher outliers. This reflects the non-deterministic nature of system-level commands that depend on filesystem state, package metadata size, and background system activity. Despite this variability, all tasks completed successfully without system crashes or inconsistent states, indicating that the Orchestrator and execution pipeline remain stable under diverse workloads.

## D. Effect of Command Output Characteristics

Figure **6** analyzes the relationship between **command output size** and observed latency.

  

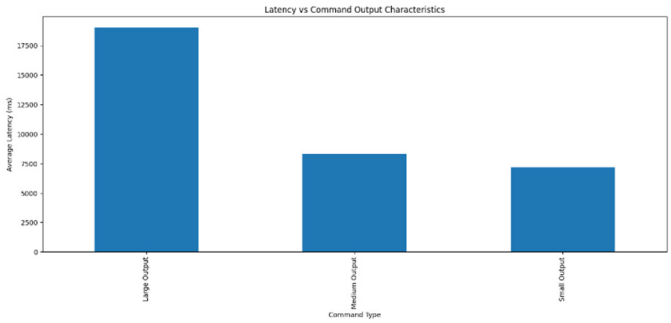


*Fig-6 Latency vs. Command Output Characteristics*

Tasks producing large volumes of output consistently exhibit higher end-to-end latency. This confirms that output handling and post-processing contribute significantly to overall execution time, often exceeding the cost of model inference itself. Tasks with small or moderate output sizes remain within interactive latency bounds, reinforcing the suitability of the system for everyday desktop operations.

### E. Discussion

The observed results demonstrate that the proposed architecture can support **practical, offline, natural-language-driven desktop automation** on consumer hardware. While latency varies depending on task complexity and output size, the system exhibits consistent behaviour and stable execution across all tested scenarios.

Importantly, the benchmarks presented in this section are **not intended as definitive performance claims**. Instead, they provide concrete evidence that a lightweight, locally deployed language model—when combined with a structured orchestration and execution framework—can effectively mediate real operating system interactions. Performance characteristics are expected to scale proportionally with system hardware and configuration, reinforcing the portability of the proposed design.

## IX.CONCLUSION

This work explored the feasibility of integrating a lightweight, locally deployed Large Language Model into a desktop operating system environment to enable natural-language-driven system interaction. By combining a compact fine-tuned language model, a centralized Orchestrator, and a minimal Debian-based base system, the proposed prototype demonstrates that meaningful OS-level automation can be achieved without reliance on cloud services.

The implementation shows that common desktop tasks—such as file management, project organization, system inspection, and development-related workflows—can be mediated through natural-language interaction while preserving user control and execution boundaries. The Orchestrator plays a central role in this design, acting as an intermediary that enforces structured parsing, validation, and controlled execution rather than granting the
language model direct access to the operating system.

Experimental observations indicate that the system operates with predictable end-to-end latency for a range of everyday tasks on consumer hardware. While absolute performance varies with task complexity and system configuration, the results provide empirical evidence that offline language models can be practically integrated into interactive desktop environments. Importantly, the evaluation emphasizes observed system behavior rather than fixed performance guarantees.

Overall, this work demonstrates the practicality of a local, privacy-preserving approach to AI-assisted desktop automation and highlights the potential of combining language-based reasoning with traditional operating system control mechanisms in a structured and transparent manner.

## X.FUTURE WORK

While the current prototype demonstrates the feasibility of offline natural-language control for desktop automation, several directions remain for future exploration and refinement.

**Voice-Based Interaction:**
Future extensions may incorporate on-device speech recognition models to enable hands-free interaction. This would allow users to issue commands and receive feedback through voice while preserving the system's local and privacy-focused design.

  

**Deeper System Awareness:**
The Orchestrator could be extended to incorporate additional system-level signals, such as I/O behavior, resource contention, or hardware health indicators. Integrating such information may enable more informed diagnostics and context-aware assistance.

**Cross-Device and Distributed Operation:**

An extension of the framework to support multiple machines—such as personal laptops, servers, or edge devices—could enable coordinated task execution and system management across distributed environments, while maintaining centralized control logic.

**Multimodal Inputs:**
Incorporating visual inputs, such as screenshots or window-level state, could enhance troubleshooting and user interaction. This would allow the assistant to reason over graphical context in addition to textual commands.

**Extensible Plugin Architecture:**
A controlled plugin mechanism could allow third-party developers to add domain-specific capabilities, such as integration with development tools or creative software, without weakening the system's execution and isolation guarantees.

**On-Device Adaptation:**
Future work may explore lightweight personalization techniques that allow the system to adapt to user preferences or recurring workflows over time. Such adaptation would need to remain fully local and operate within strict safety constraints.
These directions aim to extend the system's capabilities while preserving its core design principles: locality, transparency, controlled execution, and user authority.

## XI. XI. REFERENCES

[1] A. Vaswani et al., "Attention is all you need," in Advances in Neural Information Processing Systems, 2017, pp. 5998–6008.

[2] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in Proc. NAACL-HLT, 2019, pp. 4171–4186.

[3] T. B. Brown et al., "Language models are few-shot learners," in Advances in Neural Information Processing Systems, vol. 33, 2020, pp. 1877–1901.

[4] H. Touvron et al., "LLaMA: Open and efficient foundation language models,"
arXiv preprint arXiv:2302.13971, 2023.

[5] S. Yao et al., "ReAct: Synergizing reasoning and acting in language models,"
arXiv preprint arXiv:2210.03629, 2022.

[6] J. Wei et al., "Chain-of-thought prompting elicits reasoning in large language models," in Advances in Neural Information Processing Systems, vol. 35, 2022, pp. 24824–24837.

[7] D. Schwartz-Narbonne et al., "Formal methods for safe execution of learned control policies," IEEE Trans. Softw. Eng., vol. 48, no. 4, pp. 1243–1259, Apr. 2022.

[8] M. A. Gulzar, S. M. A. H. Rizvi, and M. A. Suleman, "Understanding and auto-fixing configuration errors in Linux systems," in Proc. ACM SIGOPS 28th Symp. Operating Syst. Princ., 2021, pp. 432–447.

[9] J. Mickens et al., "Pivot: Fast, synchronous mashup isolation using generator chains," in Proc. IEEE Symp. Secur. Privacy, 2017, pp. 261–275.

[10] B. A. Myers, A. J. Ko, and T. D. LaToza, "Natural programming languages and environments," Commun. ACM, vol. 59, no. 9, pp. 47–53, Sep. 2016.

  