Towards LLM-Native Operating Systems: Architecture, Multi-Agent Orchestration, and Real-Time Kernel Observability

Prema A[1]
Assistant professor, Department of Artificial Intelligence and Data Science, Velammal Engineering College
surapet - 600066
prema@velammal.edu.in

Siddharth Magesh[2]
Student, Department of Artificial Intelligence and Data Science, Velammal Engineering College
surapet - 600066
siddharthmagesh007@gmail.com

Deepak Kumar M[4]
Student, Department of Artificial Intelligence and Data Science, Velammal Engineering College
surapet - 600066
deepakkumarm2458@gmail.com

Beno Dharmaraaj P J[3]
Student, Department of Artificial Intelligence and Data Science, Velammal Engineering College
surapet - 600066
benodharmaraaj2905@gmail.com

Abstract — Current operating systems have not undergone any architectural changes for over six decades; the kernel operates deterministically with respect to memory, scheduling, storage, and hardware while the user learns to adapt to the language of the machine. The paper argues against such assumptions by introducing the concept of OSSARTH (Open-Source System for Autonomous Reasoning and Task Handling). It involves creating an AI-centric operating system wherein an LLM operates solely within the hardware of the system and acts as the reasoning center of the system, performing tasks such as perceiving the state of the system, planning its workloads, and controlling all kernel activities using natural language reasoning.

The OSSARTH architecture is built upon five different levels. The first level is the Base Linux Kernel, which provides hardware abstraction. The second level is the Model Context Protocol (MCP) Server Layer, where the capabilities of the system are provided as callable functions. The third layer is the Orchestration Layer, which controls the hierarchy of the Small Language Models (SLMs) from 0.5B to 7B parameters. The fourth level is the Intelligence Layer, which hosts the LLM router provided by multiple providers, with both local and cloud fallbacks. Finally, the fifth layer is the Natural Language Shell, which replaces the classic command-line shell interpreter completely.

Kernel Monitor in real time developed by utilizing Python Textual framework supports seven live observability panels on the LLM-controlled kernel, something not yet offered by any existing operating system monitoring software. The program validates that its execution is stable in concurrent processes, completes a task's lifecycle in 1.2 seconds, consumes 35 MB of idle memory, and responds to user input less than 50 milliseconds, without the use of GPU hardware.

This is illustrated by OSSARTH, whereby localizable models, through provision of structured visibility into system state as well as standardized tools, can oversee the duties that an operating system is supposed to carry out.

Keywords: LLM-Native Operating System, AI Kernel, Natural Language Shell, Model Context Protocol, Hierarchical Multi-Agent Systems, Kernel State Management, Local LLM Inference, Semantic Memory, Thread Scheduling, AI-Native Computing.

I. INTRODUCTION

1.1 Background and Motivation

Human-computer interaction is at its essence a story of concessions. Since the early days of computing, the computer dictated the rules of the engagement. Time-sharing systems in the sixties like CTSS and Multics used the command-line interface, an approach that requires a user to know the syntax of the commands and the order of the arguments, as well as flags. Graphical interfaces introduced in the eighties by the Macintosh system in 1984 and Microsoft Windows in 1985 decreased the amount of learning involved, but introduced another level of complexity through inconsistent, application-specific interaction paradigms. In both approaches, however, the underlying principle remained the same. The user had to translate intention into

machine language, and the machine performed the task. The gap between the intention of a user and a set of commands to implement the intention has not been closed, it was just hidden under the carpet.

In parallel to this, however, the operating system kernel, or the piece of software that is supposed to coordinate the use of computer hardware resources, schedule process execution, manage system memory and data storage, and maintain system stability, has developed along an equally stagnant path. From the architectural principles adopted by UNIX in the 1970s – such as hierarchies of files and file directories, isolated processes and process groups, systems call interfaces, priority-based process schedulers, and fixed algorithms for memory management – there was never any deviation, and the same principles continue to serve as the basis for all contemporary operating systems like Linux, macOS, Windows, and Android. They have proven their worth through many years of usage and experimentation. Yet, they have something in common – they are all deterministic.

The recent evolution of transformers powering large language models is arguably the biggest change in the space of computing possibilities in decades, comparable only to the introduction of graphical interfaces. Tools like GPT-4, Claude, LLaMA, and Gemini prove that a machine can understand human language, work out complicated solutions involving several steps, write functional computer code, handle file operations, navigate the internet, perform multi-phase tasks – in other words, do everything that a human operator might ask the machine to do just by describing what they would like done. Even with this remarkable power at their disposal, however, LLMs have been mostly limited to stand-alone chatbots and API integrations. They exist on a level above the OS and serve as advisors, not agents. The operating system they work on relies on its deterministic kernel as always, unaware of the smarts sitting atop of it.

The central question this paper asks is: what would change if the LLM was not sitting on top of the operating system, but was the operating system's governing intelligence?

1.2 The Problem with Existing Systems

The provided text argues that current operating systems suffer from four fundamental structural flaws that misalign machine functionality with human cognition. First, command-line interfaces demand an unnatural, rigid syntax to execute straightforward human intentions. Second, graphical user interfaces are highly fragmented, forcing users to learn completely distinct interaction patterns for different applications, which especially complicates system administration. Third, a high technical barrier prevents non-experts from executing complex tasks like server configuration or process management. Finally, traditional OS kernels lack intelligent resource management for modern AI workloads; standard monitoring tools only track basic hardware usage and cannot comprehend or prioritize the specific semantic needs of AI tasks, such as model weights, inference buffers, or user intent.

1.3 The Vision: LLM as the Kernel Intelligence

OSSARTH introduces a paradigm shift by utilizing a Large Language Model (LLM) not merely as an agent operating within a traditional OS, but as the core reasoning engine that actively governs the OS kernel itself. Unlike systems such as AutoGPT or OS-Copilot that rely on deterministic kernels to decide how resources are allocated, OSSARTH empowers the LLM to direct computing resources using structured system visibility and standard interfaces. This novel approach is made feasible by three converging technological trends: the rise of accessible, locally deployable LLMs via tools like Ollama that can run on consumer hardware without GPUs; the standardization of LLM-tool interactions through Anthropic's Model Context Protocol (MCP); and the advancement of Python-based programming frameworks like Textual for building integrated terminal dashboards.

1.4 What OSSARTH Does

The OSSARTH system realizes this vision as a fully functional implementation of an AI-native operating system architecture consisting of five layers. In the lowest level is a traditional Linux kernel that abstracts the hardware and supplies only basic primitives to the system. The layer above is a Model Context Protocol (MCP) server layer, which offers the capabilities of the system as standardized tool interfaces based on schemas that the LLM can call based on its reasoning of the required system functionalities. The functions offered by these MCP servers include file system management, Git version control, web fetching, and memory persistence.

Above the MCP module, an orchestration module is responsible for decomposing complex jobs and distributing them through a hierarchy of intelligent layers. A 7-billion parameter master orchestration model performs reasoning at a higher level, analyzes intentions, and performs multi-stage plans. Below this layer, there is a set of small language models of various sizes from 0.5 to 3 billion parameters. These models perform various specific tasks such as code generation for programming, a retrieval model for searching and looking up information, a memory model for context window management, a planning model for multi-stage workflow decomposition, and an embedding encoder for semantic indexing. Three smaller lightweight

LoRA adapters add capabilities for summarization, classification, and translation without the additional memory consumption associated with large models.

The intelligence layer hosts a multiprovider LLM router that keeps the system available using failover to other local inference engines such as Ollama and cloud inference engines like Groq, Gemini, and OpenRouter, thus ensuring that the governing intelligence of the OS stays up even if some inference engines are down. On top of the stack is a Natural Language Shell, which acts as an alternative to the command-line interface that understands natural language intents, executes them, and lets us know how it went as well as which tools, agents, and rationale it used.

All of these systems run atop of the Kernel State Manager, which is a thread-safe singleton that holds the comprehensive up-to-date state of all AI kernel subsystems at one time. The state includes hardware monitoring information regarding CPU, GPU, network, and disk, memory usage statistics across eight semantic domains of kernel tasks, model weights, task buffers, inference buffers, token buffers, embedding caches, vector indices, and system caches, a model registry containing twelve running model instances and information about them including hierarchy, status, and inference number, a thread pool of sixty-four threads with individual thread state management, a task queue with twenty-five sample AI tasks' life cycle management, a semantic indexed file storage system, and a structured kernel logs stream. Any change of state data is guarded by a reentrant lock.

In order to bring the governance of such systems by the LLM to light and make it understandable, OSSARTH presents the Kernel Monitor. It is a real-time terminal dashboard developed utilizing the Python Textual library. The seven-page observability software shows live data regarding the system dashboard, kernel logging, model registry, memory management, semantic storage, thread scheduling, and hardware monitoring, updated at every five hundred milliseconds with global keyboard control. Currently, there are no other OS monitoring applications that offer this feature since there are no other OS monitoring applications that can be used for such a system.

1.5 Contributions of This Work

This paper makes five key contributions to the intersection of operating systems, AI, and human-computer interaction. First, it introduces a modular, five-layer reference architecture separating hardware, tools, orchestration, LLM intelligence, and natural language interfaces to enable the independent evolution of each component. Second, it proves the practical feasibility of LLM-driven kernel subsystems by successfully managing memory, scheduling, storage, and tasks on local hardware without requiring a GPU. Third, it adopts the Model Context Protocol (MCP) as a standardized integration bus, establishing an extensible ecosystem by solving the complex N×M tool compatibility issue. Fourth, it implements a hierarchical multi-agent delegation structure that utilizes multiple smaller models rather than one massive model to maximize both decision-making accuracy and computational efficiency. Finally, it introduces the Kernel Monitor, a novel real-time observability tool designed specifically to interpret the semantics of AI-native operations, successfully filling a blind spot left by traditional deterministic system monitors.

1.6 Paper Organization

This paper will proceed as follows. In section 2, a review of pertinent studies on LLM-based OSs, Autonomous Agents, Model Context Protocol, and terminal interface framework is presented to pinpoint the deficiencies that OSSARTH overcomes. Section 3 will describe the architecture of the system with detailed discussion about each layer. Section 4 will discuss the approach used during design, implementation, and evaluation. Section 5 will go into details about how each system component is implemented. Section 6 will present experimental results and evaluate them regarding performance and bugs. Section 7 will discuss the challenges encountered during prototype development. The conclusion will be drawn in section 8, and the direction of future research in section 9.

II. LITERATURE SURVEY

2.1 Overview

OSSARTH's design is based on a rich and constantly growing literature involving LLM-powered operating systems, autonomous agents, memory management techniques for language models, tooling interoperability standards, terminal interfaces, and local inference optimization. In this section, we critically analyze the seminal papers in each of these categories, discuss their limitations, and show how OSSARTH overcomes these limitations using its own architecture.

2.2 LLM-Based Operating Systems and Agent Kernels

The foundational lineage of the AIOS framework spanning the conceptual "AI Kernel" introduced in AIOS v2 (Mei et al., 2024), the concurrent sub-execution pipelines optimized for efficiency in AIOS v5 (Mei et al., 2025), and the asynchronous, microkernel-style architecture revealed in its open-source codebase has directly shaped the core architecture of OSSARTH. While AIOS established critical mechanisms for scheduling LLM agents and managing shared systemic resources like context

windows, storage, and memory, it suffered from several significant limitations: a strict dependency on high-end multi-GPU server infrastructure, proprietary tool interfaces, centralized scheduling bottlenecks, and discrepancies between theoretical claims and actual codebase implementations. OSSARTH resolves these foundational shortcomings by adapting AIOS's core agent-serving theories into a robust framework engineered specifically for standard commodity hardware, rather than specialized enterprise multi-GPU environments. Furthermore, instead of relying on a fragmented and proprietary ecosystem, OSSARTH adopts the vendor-agnostic Model Context Protocol (MCP) to seamlessly resolve tool compatibility (the N+M scenario), while concurrently introducing real-time state monitoring and a decomposed Task Execution Engine based on AIOS v5's pipeline to execute agent sub-tasks with high concurrent efficiency.

2.3 Memory Management for Language Models

2.3.1 MemGPT: Towards LLMs as Operating Systems (Packer et al., UC Berkeley, 2023)

Packer et al.'s MemGPT introduces a core concept analogous to OSSARTH's memory approach: a language model's context window acts like limited computer RAM, necessitating a "virtual memory" system that pages data between RAM and disk as needed. To manage this capacity, MemGPT employs a dedicated memory agent that moves data across three distinct tiers based on recency and relevance: a main context (acting as working memory or RAM), an external memory (a vector database acting as a hard drive), and a working context for the current task.

This hierarchical structure directly inspired OSSARTH's architecture, specifically its use of a dedicated memory management agent and its eight-category memory allocation scheme, which mirrors MemGPT's main and external storage through fast inference/token buffers and embedding caches. However, OSSARTH advances beyond MemGPT by integrating these features directly at the operating system level; while MemGPT operates as a standalone agent, OSSARTH provides kernel-level memory state observation, multi-agent memory coordination, and active telemetry via its Kernel Monitor.

2.4 Autonomous Agents and Tool Use

2.4.1 ReAct: Synergizing Reasoning and Acting in Language Models (Yao et al., Princeton/Google, 2022)

The ReAct approach (arXiv:2210.03629) was first presented by Yao et al. at ICLR 2023 and defined the Thought-Action-Observation sequence as the basis for LLM-powered agents. According to the ReAct model, the agent generates a reasoning chain expressed in natural language (Thought), calls upon a tool or performs an action (Action), observes the outcome of the action performed (Observation), and repeats this sequence of reasoning and actions until the task is accomplished. Yao et al. demonstrated that the ReAct approach greatly surpasses simple chain-of-thought reasoning on HotpotQA, FEVER, and AlfWorld datasets.

The ReAct approach itself is literally hard-coded into OSSARTH's orchestration framework. The master orchestrator LLM operates exactly according to this cycle: it analyzes the intention of the user and the current state of the system (Thought), calls the relevant MCP tool or sub-agents (Action), gets back the structured tool output or agent response (Observation), and keeps iterating until the task is completed, resulting in a state change in the system to a new desired state. The TaskEngine execution phase log containing information on how each phase progresses with detailed explanations of the operations performed by the model is exactly a concrete realization of the thought trace in ReAct.

2.4.2 Toolformer: Language Models Can Teach Themselves to Use Tools (Schick et al., Meta AI, 2023)

In their paper (arXiv:2302.04761), Schick et al. have proven that a model with 6.7 billion parameters using structured tools is significantly better at solving various tasks than a GPT-3 with 175 billion parameters. In other words, having a more complex algorithmic structure for model interaction with its environment allows a smaller and more computationally efficient model to solve problems that are usually solved by orders of magnitude larger models.

The above result proves the necessity of the OSSARTH's hierarchical delegation scheme. Instead of training one large 70B+ model to perform all tasks assigned to the operating system, OSSARTH employs a 7B orchestrator model to reason about tasks and a number of specialized sub-models with sizes varying from 0.5B to 3B each. Each of the specialized agents gets structured access to tools through MCP. Structured tool use is replicated in an operating system setting in accordance with the Toolformer paradigm. The resulting computational cost is significantly lower since a sum of parameters across all twelve models that OSSARTH uses is estimated to be around 9.38B.

2.4.3 OS-Copilot: Towards Generalist Computer Agents with Self-Improvement (Wu et al., 2024)

The work of Wu et al. (arXiv:2402.07456) introduces the system of OS-Copilot, which allows interfacing between an LLM and a general-purpose computer using a standardized tool library. The primary contribution of OS-Copilot is showing that LLM-based agents are able to perform complex OS-related tasks such as file operations, process handling, shell command

execution, and graphical user interface manipulations provided that the tools offered for interfacing with an OS have consistent, well-defined interfaces.

In terms of tool concepts, OS-Copilot confirms the effectiveness of OSSARTH's MCP tooling architecture. Like the OSSARTH system, OS-Copilot acknowledges the need for structured interfaces described with schemas rather than generating random shell commands for an LLM-based agent. Unlike OSSARTH, OS-Copilot is implemented as a layer over a classical operating system rather than an LLM-based OS replacement. OS-Copilot is also based on a custom tool library rather than using a standardized protocol like OSSARTH.

2.5 Model Context Protocol

2.5.1 Introducing the Model Context Protocol (Anthropic, November 2024)

The Model Context Protocol, introduced by Anthropic in November 2024, is an open protocol for interfacing AI models to their external environment via client-server architecture. MCP defines three categories of capabilities that servers may offer: Tools, callable methods that correspond to system calls; Resources, readable data sources similar to file descriptors; and Prompts, predefined input templates for various interaction styles. For transport, MCP allows the use of stdio for communication with local subprocesses and HTTP/Server-Sent Events for remote communication.

For an LLM-OS implementation, the architectural significance of MCP lies in its potential to turn tool integration into an N+M rather than an N×M task, allowing any MCP client to utilize any MCP server without requiring a unique integration solution for each LLM provider and each category of capabilities. By early 2026, the ecosystem of MCP servers already includes hundreds of official servers and thousands of community servers providing database, cloud, developer tooling, version control, productivity application, and hardware interfaces.
For OSSARTH, the only communication bus used by the intelligence layer for all the capability servers is MCP. There are three reasons for this choice. First, OSSARTH becomes instantly interoperable with any current MCP server; second, any additions to the system capabilities do not need modifications to be made to the LLM or orchestration layers; and third, OSSARTH takes advantage of the evolving MCP ecosystem without doing any programming.

2.6 Local LLM Inference Frameworks

The local execution paradigm of OSSARTH relies on an optimized inference stack that eliminates external API dependencies and minimizes hardware constraints. Ollama serves as the primary inference backend, providing an offline REST API structured compatibly with OpenAI's message format, which enables seamless switching via OSSARTH's LLM router. Under the hood, this backend leverages the capabilities of llama.cpp to perform efficient model quantization (such as $INT4$ and $INT8$), reducing memory footprints by 2x to 4x and allowing a 7B parameter orchestrator model to run efficiently on consumer-grade CPU hardware without requiring a dedicated GPU. Furthermore, the memory architecture of OSSARTH draws a direct conceptual parallel to vLLM's PagedAttention mechanism; by treating the Key-Value (KV) cache as non-contiguous, virtual memory pages, OSSARTH adopts a similar virtual paging metaphor to orchestrate and manage system-level agent memory allocation dynamically.

III. System Architecture

3.1 Overview

OSSARTH OS Architecture is built as a vertically integrated architecture of OS Architecture that consists of different layers, where Large Language Model serves as a reasoning engine for the whole system. Unlike in a deterministic architecture, where decisions regarding the actions of an operating system are made by a deterministic kernel using deterministic algorithms, OSSARTH leaves this responsibility to the model hierarchy because it has the ability to make decisions based on the knowledge about system state and workload. The architecture consists of seven layers.

Layering ensures that no layer spans across layers. The NL Shell does not directly call any of the MCP Servers. The MCP servers do not directly call model inference. The Linux Kernel does not directly receive inputs of natural language. All interaction between layers is achieved through the interface provided by the layer just beneath it, thus enabling replacement of any layer individually without affecting others.

The complete system architecture from user input to hardware execution is shown below:

USER INTERFACE LAYER

Voice / Text / Multimodal Interaction

↓ Raw Input (Voice / Text / File) ↓

LAYER 5 | SPEECH PROCESSING LAYER

Wake-Word Detection	Automatic Speech Recognition	Speaker Identification	Noise Reduction
Always-on Activation Trigger	ASR: Audio -> Text Transcription	Biometric Voice Authentication	Signal Enhancement Pre-NLU Processing

↓ Text Query ↓

LAYER 4 | NATURAL LANGUAGE UNDERSTANDING (NLU)

Intent Recognition	Entity Extraction	Context Tracking	Complexity Classification	Modality Classification	Sensitivity Classification
What does user want?	Files, paths, params	16K Sliding Window	Simple / Multi-step / Reasoning-heavy	Text / Code / File / System Operation	Safe / Privileged / Destructive / Irreversible

↓ Structured Query ↓

LAYER 3 | COGNITIVE CORE (LLM ENGINE)

MASTER ORCHESTRATOR (7B params)

- LLM Reasoning Engine: ReAct: Thought -> Action -> Observation
- Task Planning: Decomposes complex intent into subtasks
- Tool Selection: Decides which MCP tool/agent to invoke
- Multi-step Decision Making: Chains reasoning across multiple tool calls
- RAG Integration: Retrieves relevant context from vector store
- [LLM ROUTER (Provider Failover)]
- Ollama (Local) <-> Groq (Cloud) <-> Gemini

SPECIALIZED AGENTS

- Code Agent (1.5B): Code Generation, Debugging, Review
- Retrieval Agent (1B): Search, Lookup, Retrieval
- Memory Agent (0.5B): Context Window, Compression, Mem Mgmt
- Planning Agent (1B): Workflow Planning, Roadmap

REASONING ENGINE (3B params)

- Summarization (LoRA: 50M): Text Compression
- Classification (LoRA: 50M): Intent Tagging
- Translation (LoRA: 80M): Multilingual Support
- RAG & Vector Store: Semantic Search, Knowledge Retrieval

↓ MCP Tool Call (JSON-RPC over stdio) ↓

LAYER 2 | MCP SERVER LAYER

Tool Servers

- Filesystem MCP: read, write, list
- Git MCP: commit, diff, log, branch
- Web Fetch MCP: fetch_url, search_web, parse_html

Memory MCP

- store_memory
- retrieve_memory
- update_memory
- forget_memory
- Backend: ChromaDB / Qdrant

Security Manager

- Permission Level Check
- Confirmation Prompt
- Audit Logging
- Rate Limiting

↓ Syscall / Kernel Operation ↓

LAYER 1 | AI KERNEL MANAGEMENT LAYER

Kernel State Manager

- Hardware Monitoring
- Memory Management
- Model Registry (12 inst.)
- Thread Pool (64 max)
- Task Lifecycle (6 states)

Task Execution Engine

- QUEUED
- SCHEDULED (0.4s)
- ALLOCATING (0.2s)
- RUNNING (0-100%)
- COMPLETED / FAILED

Memory Allocation Map

- Kernel Processes: 1,024 MB
- System Cache: 512 MB
- Model Weights: Dynamic
- Task Buffers: 45%
- Inference Buffers: 35%
- Token Buffers: 20%

System Monitoring

- CPU / GPU Usage
- Memory Usage
- Network I/O
- Disk I/O
- Real-time Metrics

↓ ↓

LAYER 0 | BASE LINUX KERNEL

Hardware Abstraction	Process/Thread Mgmt	Virtual Memory	Filesystem (ext4, tmpfs)	Device Drivers

3.2 Layer-by-Layer Description

Layer 0 -- Base Linux Kernel

Base Linux Kernel is the sole component that works using deterministic principles. It abstracts hardware, manages processes, performs virtual memory paging, mounts file systems, and implements device drivers. In OSSARTH, this layer is downgraded from its conventional position as the ultimate decision maker for users to a reliable platform for executing tasks. It does not get commands from users but gets directions from the AI Kernel Management Layer above it. Users never make direct system calls to the kernel. Each function performed by the Linux kernel has gone through logical reasoning and approval in the LLM layers above it.

Layer 1 -- AI Kernel Management Layer

This is the most architecturally innovative layer in OSSARTH and the main innovation of the system. The layer includes three key modules that take on all responsibilities typically fulfilled by a deterministic kernel.

The Kernel State Manager is a thread-safe singleton that is guarded with the threading.RLock. It holds the full state of all AI kernel subsystems concurrently at any point of time: telemetry data about hardware state updates in every 500 milliseconds, eight-memory categories accounting for 32,768 megabytes of RAM, twelve-model registry with hierarchical relations and states, a thread pool of 64 threads, including the state of each of them, task queue and manager of their life cycle, a semantically-indexed system of file storage holding 500 gigabytes and a ring buffer log of 1,000 items. Each state mutation involves locking acquisition. The subsystems use the same singleton state.

The Task Execution Engine executes as a daemon thread that polls the task queue at every 500ms. Upon dequeuing of a task, a new task-specific daemon thread is created for the execution of the task in the six states. During the ALLOCATING state, memory is explicitly allocated based on three different buffers, which take up 45%, 35%, and 20% of memory respectively. During the RUNNING state, each task goes through various stages, while producing log events and updating hardware telemetry upon each of those steps. During the COMPLETING state, all allocated memory is released using the deallocate_memory() function, guaranteeing that no memory leaks occur during task completions.

Memory Allocation Map breaks down the total memory into eight categories that represent the actual use cases of AI algorithms instead of measuring the amount of memory consumed by process ID in terms of bytes. Two out of the eight categories are constant and unchanging: kernel processes with 1,024 MB and system cache with 512 MB. Six other categories dynamically change after each task starts or finishes.

Layer 2 -- MCP Server Layer

MCP Server Layer
Capabilities provided by the system are surfaced as standardized tool interfaces according to a defined schema, which the LLM then calls using an agreed-upon signature and typed arguments. The LLM does not produce raw shell commands. All interaction with the system goes through a named, audited, and permission-controlled tool call from the MCP, over JSON-RPC with stdio transport.

Four MCP servers are currently active in the current architecture. Filesystem MCP is responsible for providing full file manipulation capabilities: reading, writing, listing, deleting, and searching for files within the local filesystem. Git MCP provides version control capabilities: commit, diff, log, status, and branch manipulation. Web Fetch MCP provides internet capability: fetching web pages via URLs, web search, parsing HTML, and extraction of data. Lastly, Memory MCP provides semantic memory persistence: creation, retrieval, updating, and forgetting of agent memories with support from ChromaDB or Qdrant vector databases.

All requests made by the tools go through the Security Manager before being executed. The Security Manager checks each request against a policy of permissions, adds a manual confirmation request for each potentially destructive task, keeps a detailed audit log with timestamps, and imposes rate limiting in order to keep agent actions in check. This makes sure that all the actions undertaken by the LLM are safe and easily auditable.

The major strength of the MCP design lies in its extensibility. In order to provide a new functionality in the system, it is enough to introduce a new compatible server and register it with the orchestrator. None of the LLMs and no router needs to be changed at all. This is the N+M approach to the N×M problem.

Layer 3 -- Cognitive Core (LLM Engine)
Cognitive Core is the place where all intelligence of OSSARTH resides. It comprises three hierarchical layers of reasoning capabilities.

Master Orchestrator is the main reasoning authority. It consists of 7B parameters and performs tasks through the Reasoning Loop based on the ReAct protocol, i.e., Thought (reasons about intent), Action (calls upon a tool/sub-agent), Observation (processes the response), and further repeats until completion. It formulates a workflow of actions to achieve a goal, determines which MCPs are needed to complete it and calls upon the corresponding sub-agents to execute particular tasks.

There are five sub-agents responsible for executing particular specialized tasks: code generation, debugging, code review, search, lookup, and information retrieval. The code agent consists of 1.5B parameters, and the retrieval agent of 1B parameters. Both the code agent and the retrieval agent perform their particular functions. There is a 0.5B parameter memory agent that manages context window use and intelligently compresses information to overcome token limitations. The planning agent consisting of 1B parameters decomposes multi-step tasks into simpler actions. Finally, the embedding encoder of 350M parameters generates embeddings for semantic indexing.

Reasoning Engine at 3B parameters is used for issues that require a greater level of logical thinking and is enhanced with three LoRA adapters – the summarization adapter with 50M parameters, classification adapter with 50M parameters, and translation adapter with 80M parameters. These adapters can easily be loaded into the underlying reasoning engine with no significant increase in memory requirements to give specialized output results without maintaining separate instances of the model.

LLM Router ensures the system availability with automatic failover to five provider backends, in order of priority - Ollama for local inference, Groq Cloud, Gemini, OpenRouter, and HuggingFace. Each of the providers supports the common interface of BaseLLMProvider and implements such functions as chat(), stream(), and health_check().

Layer 4 -- Natural Language Understanding (NLU)

NLU layer turns plain text inputs from the users into well-formed queries understood by the Cognitive Core for executing reliably. It is built on top of three classifiers: complexity classifier, which distinguishes the nature of request as simple Q&A, tool execution, workflow execution, and reasoning; modality classifier, which understands text generation, code, file manipulation, and system actions; sensitivity classifier, which highlights dangerous requests that require destructive/irreversible privileged operation execution.

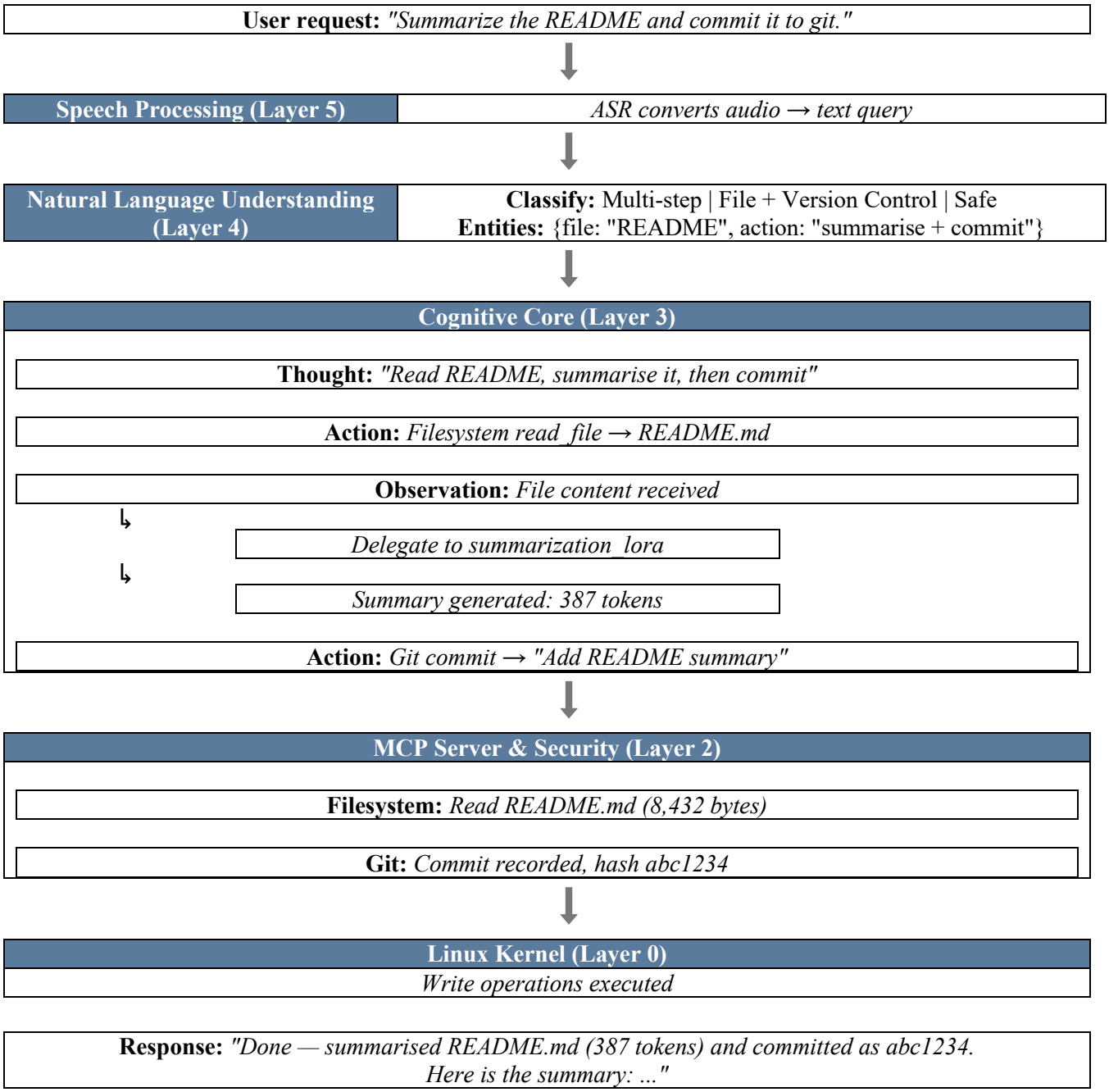
Context manager within NLU layer tracks conversational history of 16K tokens sliding window with prioritization of queuing. Pinning allows retaining pinned messages regardless of other constraints. Budget allows storing recent messages until the budget ends. Semantically related past messages are fetched through cosine similarity from previous messages relative to the current message.

Layer 5 -- Speech Processing Layer

Speech processing is provided by the Speech Processing Layer, which allows voice as an input mechanism on par with other modes of interaction. Wake-word detection makes sure that the system stays in a low-power state until the keyword is heard and activates the processing. The automatic speech recognizer converts audio data into text using local inference, which does not rely on cloud services. Speaker identification and speaker authentication authenticate the speaker by identifying their voice. Noise cancellation and signal enhancement techniques help in improving the quality of transcription in noisy conditions. The output of this layer is processed text, which is then sent to the NLU layer just like the typed input.

3.3 Cross-Layer Interaction: Complete Request Flow

The following diagram shows the precise sequence of events when a user issues a compound command "summarize the README and commit it to git" tracing the request from voice input through every layer to hardware execution and back:



Total Latency: **~3.2s** | Model: Local Ollama | No Cloud API Used

KernelState updated at every step | Full Audit Log Recorded

3.4 Model Registry

The following table describes all twelve model instances maintained in the Kernel State Manager's model registry, their positions in the agent hierarchy, and their resource profiles:

ID	Name	Type	Parent	Mem (MB)	Params	Layers
mdl-001	**master_orchestrator**	master	-	4,200	7B	32
mdl-002	**reasoning_engine**	master	-	1,800	3B	24
mdl-003	**code_agent**	-	mdl-001	900	1.5B	16
mdl-004	**retrieval_agent**	sub_agent	mdl-001	900	1B	16
mdl-005	**memory_agent**	sub_agent	mdl-001	600	0.5B	12
mdl-006	**planning_agent**	sub_agent	mdl-001	600	1B	8

mdl-007	**summarization_lora**	adapter	-	50	50M	4
mdl-008	**classification_lora**	adapter	-	50	50M	4
mdl-009	**translation_adapter**	-	mdl-002	80	80M	4
mdl-010	**math_expert**	adapter	-	500	80M	6
mdl-002	**science_expert**	expert	mdl-002	500	500M	10
mdl-011	**embedding_encoder**	-	mdl-002	350	350M	6
Total model weight memory (all active): ~9,930 MB.						

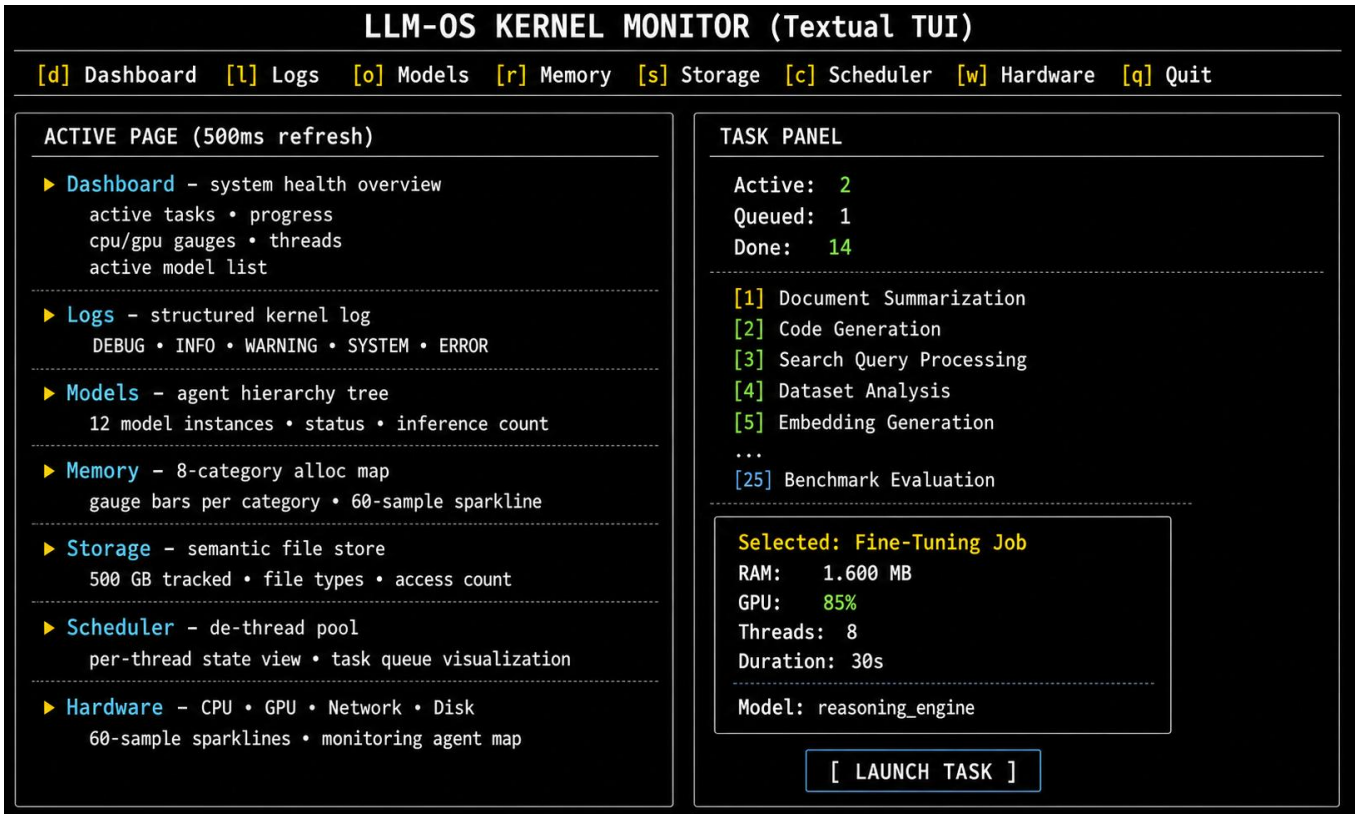
3.5 Task Definitions Summary

Twenty-five AI workload types are defined in the system, each with a fully parameterized resource profile. The following table summarizes their resource characteristics:

#	Task Name	Duration	RAM (MB)	GPU %	Threads	Phases
01	Document Summarization	12s	420	35%	4	5
02	Code Generation	15s	680	45%	6	6
03	Search Query Processing	8s	320	20%	3	4
04	Dataset Analysis	18s	850	30%	8	6
05	Embedding Generation	10s	520	55%	4	5
06	Document Indexing	14s	600	25%	5	6
07	Multi-Agent Reasoning	22s	1,200	60%	7	6
08	RAG Query Pipeline	11s	550	40%	5	6
09	Conversation Inference	10s	320	20%	3	4
10	System Diagnostics	7s	180	20%	6	6
11	Data Ingestion Pipeline	18s	950	30%	8	6
12	Knowledge Graph Building	12s	1,100	40%	7	7
13	Distributed Inference	16s	1,400	70%	5	6
14	Sentiment Analysis	10s	350	25%	4	5
15	Code Review & Analysis	14s	580	35%	5	6
16	Image Caption Generation	11s	750	30%	5	5
17	Translation Pipeline	10s	1,100	40%	7	7
18	Question Answering	9s	380	25%	4	5
19	Text Classification	10s	420	25%	4	6
20	Anomaly Detection	12s	400	25%	5	4
21	Recommendation Engine	10s	700	35%	6	6
22	Chatbot Session	14s	550	35%	4	6
23	Fine-Tuning Job	30s	1,600	85%	8	8
24	Model Compression	13s	1,600	50%	6	6
25	Benchmark Evaluation	25s	1,000	60%	10	7

3.6 Kernel Monitor -- Observability Layer

The Kernel Monitor is a real-time terminal dashboard that makes the LLM's governance of all kernel subsystems visible and interpretable to the operator. It is built on the Python Textual 8.1.1 framework and presents seven monitoring pages, all connected to the KernelState singleton and refreshed at 500ms intervals.



The Monitor reads exclusively from the KernelState singleton and never writes to it. It is a pure observer it never influences system behavior. This clean separation ensures that the monitoring layer can never cause side effects in the kernel it is observing, and that the kernel operates identically whether or not the monitor is running.

IV. Methodology

4.1 Overview

The development methodology employed for OSSARTH can be best described as an architectural-driven, documentation-based, iterative software engineering process broken down into three consecutive development stages. All phases depended upon the results from the prior phase, meaning that all implementation steps taken had an architectural basis behind them and that all components implemented were independently verified. This specific choice was driven by the desire to make the work as methodologically rigorous as possible and therefore replicateable, following the spirit of systems research and moving away from an ad hoc iterative nature typically associated with software prototypes.

These three phases include: Phase 1 – System Architecture Design & Documentation; Phase 2 – Backend Infrastructure Development & MAS Implementation; Phase 3 – Kernel Visualization, State Management & Verification.

4.2 Phase 1 -- Architecture-First Design

4.2.1 Philosophy: Documentation Before Code

The primary, indeed, most critical, methodological decision made in this project was to conduct a thorough architectural design prior to even touching the code for application development. The rationale for doing so was that what fails with virtually all prototypes of the LLM-OS concept is not the model itself, but its integration into the system due to the lack of architectural rigor involved in the integration process. Indeed, the projects such as AutoGPT and BabyAGI crashed on their own weight mainly due to the lack of architectural design that resulted in their ad hoc growth. Therefore, OSSARTH was intended to prevent such an eventuality by prioritizing architectural design.

Phase 1 yielded twelve documents, comprising about fifteen thousand words altogether, covering all the key design decisions in the project, from the selection of the base operating system to the design of individual MCP schemas.

4.2.2 Technology Stack Evaluation

Technology stack selection was performed using a decision matrix approach where potential components were rated based on a pre-defined set of criteria per component class. The following criteria were considered when choosing the base operating system: hardware compatibility, maturity of the package ecosystem, overall system stability, easy customization for building live ISOs, wide community support, small installation footprint, and maintainability. Debian was chosen as the best choice owing to its unsurpassed stability record, wide hardware compatibility, and mature debootstrap toolchain for producing customized ISO images.

When selecting an LLM inference backend, Ollama was chosen as the main local inference service provider ahead of alternatives such as llama.cpp directly, LM Studio, and Jan. This choice was driven by the ability of Ollama to support OpenAI's message format via a REST API, provide support for quantized GGUF model types across many model families, support daemon mode, which is convenient for integration into always-on systems, and allow easy switching between providers via the abstraction of OSSARTH's router component. vLLM was considered for its efficient PagedAttention implementation but excluded from the main backend due to its mandatory GPU dependency.

The chosen terminal user interface library is Textual, compared to urwid, curses, and blessed, because of the CSS-like layout system, the reactive state management mechanism similar to those in contemporary frontend frameworks, and importantly, the run_test() API used for headless testing of the application programatically, which was crucial from the early stages of the project design.

4.2.3 Five-Layer Architecture Design

The five-tier structure was devised based on an iterative approach, whereby classical OS abstractions were mapped to LLM counterparts using the theoretical constructs developed in AIOS, MemGPT, and the Illustrated LLM OS. The specification of each layer included three parts: a description of its sole responsibility, input specifications that detail what it receives from the tier above it, and output specifications that state what it outputs to the tier below it. This specification framework was strictly adhered to, ensuring that no layer took up the responsibility of any other layer.

MCP layer is one of the most important layers that have been designed with special consideration since it is the most innovative part of the architecture. In the designing of the MCP layer, all of the more than thirty MCP servers from the ecosystem community have been identified, classified based on their capabilities, and only four of the most important ones were selected, including filesystem, git, web fetch, and memory. The use of stdio over subprocess pipes was preferred to HTTP-SSE because of low latency and simplicity.

4.2.4 Multi-Agent Hierarchy Design

The agent hierarchy was created by first listing all possible OS-level tasks that OSSARTH had to accomplish, grouping those tasks according to the type of reasoning required to accomplish them, and allocating a model size to each group that reflects the minimum number of parameters necessary to accomplish the particular form of reasoning with a high degree of accuracy. Informed by the Toolformer results showing that structured models with awareness of tools have better task-specific performance when they have fewer parameters (6.7B) compared to unstructured models with no tooling (175B), and empirical data from the Ollama benchmark suggesting that 1B-3B models can perform adequately within specific domains using only 2GB of memory, the agent hierarchy adopts a bottom-up approach toward estimating the number of parameters in each model.

The choice of LoRA adapters for summarization, classification, and translation tasks instead of individual models was made because of the memory-saving advantage offered by LoRA adapters. The use of a LoRA adapter with 50-80MB takes the place of having an extra model with 300-900MB.

4.3 Backend Infrastructure and Multi-Agent Implementation

4.3.1 Component-by-Component Implementation Strategy

Phase 2 took the approach of implementing each of the backend components individually until they all were ready to be integrated. This ensured that the integration would not become the bottleneck due to the typical problem of integration time debugging, where bugs within one of the modules create misleading errors within other modules, making it very hard to identify the root cause.

The following five components were implemented in the sequence listed below.

4.3.2 LLM Router Implementation

The LLM Router came up first since it's the layer that other intelligence components depend on. It provides an abstraction of the provider differences via an interface with three methods – synchronous chat for request-reply calls, a stream call for output delivery by tokens, and the health method for checking if a provider is available. All providers implement the same interface separately, allowing switching between providers without changing anything in the orchestration layer.

For implementing the failover strategy, we've used a prioritized list of providers. Every time when a request for inference comes up, the router tries the highest priority provider. In case of an error, timeout, or unhealthy status after health method call, the next available provider is selected from the list without informing the caller about an error. There is also the failover logic implemented that keeps track of the number of errors and uses exponential backoff on the failed providers.

Adapters were developed for Ollama using the REST API available in Ollama, Groq, and Gemini using their SDKs in Python, and OpenRouter and HuggingFace using the message format available in OpenAI. In all cases, the cloud provider adapters retrieve the credentials through environment variables; hence, the system will run offline when there are no credentials available, making use of only the Ollama local adapter.

4.3.3 Task Classifier Implementation

The task classifier was developed using a multi-stage sequential pipeline approach applied to all incoming requests made by users. The initial stage of the pipeline was the complexity classifier which passed the request by the user through a light-weighted SLM with a structured prompt and returned one of the five specified complexity levels. The classification output is not free-form but structured in the JSON format to ensure deterministic routing irrespective of how verbose is the generated response by the model.

The application of a structured JSON output rather than free-text parsing was a result of a conscious decision to make sure that this part of the pipeline will be robust enough for real-world applications. Classification outputs generated by a free-form text parser vary depending on the version of the model, inference temperature, and even slight alterations to the prompt. This problem is eliminated using the JSON format.

4.3.4 Context Manager Implementation

Context manager was designed as a solution to one of the primary problems, which is keeping coherent mult-turn dialogues within a limited number of tokens. The module stores the whole history of messages and provides the prioritized truncation mechanism once the total length of stored history approaches the limitation on the number of tokens that can be processed. There are three levels of priority assigned to different messages: pinned messages should always remain in the history as the most important instructions from the system; recent messages are included in reverse chronological order until the remaining budget runs out; messages that are included due to semantic similarity with the current query are called retrieved messages.

A priority-based queueing system was selected as a solution to the problem instead of simple methods like removing the oldest messages or generating a summary for the whole history since priority-based approach guarantees that all messages relevant for a particular query would remain in the active context window. If a user asks another question concerning the information provided by the chatbot several turns ago, the response will contain a reference to the previous message since it is semantically similar to the current one.

4.3.5 MCP Orchestrator Implementation

The MCP Orchestrator is responsible for handling the whole lifecycle of external MCP servers processes. At the startup, it initializes each defined MCP server as a subprocess and sets up stdio channels via the client part of the MCP Python SDK implementation. Tool discovery functionality consists of collecting a list of tools provided by each of the MCP servers and creating an internal tool registry where the names of the tools are associated with servers and schemas. While running, it forwards all tool calls to the right server subprocess, performs JSON-RPC serialization and deserialization, and merges tool responses to include them in the conversation context for further reasoning of the LLM.

In addition to forwarding calls, the orchestrator runs a health monitor loop that regularly sends pings to each of the subprocesses and tries to automatically restart them if there is no response for a certain amount of time. Such a feature allows avoiding disabling of any system capability because of a dead MCP server during one session period.

4.3.6 Security Manager Implementation

The security manager was designed to act as an intercepter that wraps every single tool call prior to passing to the MCP orchestration component. The security manager uses the permission matrix that categorizes all the existing tools into three groups: tools for executing safe operations which don't require any further authorization, tools for executing elevated operations which must be logged but not authorized, and tools for executing destructive actions which should be authorized by users prior to their usage.

Each tool call intercepted by the security manager generates a record in an audit log that contains the date and time when the operation has been performed, an agent that executed the operation, the name of a tool, the set of arguments provided to the tool, and the results of tool execution.

4.4 Kernel Visualization and State Management

4.4.1 Strict Decoupling Strategy: State Before UI

Phase 3 used a strict decoupling approach where the state management layer would be developed and tested independently before any coding work on the user interface layer begins. The rationale for using this approach is based on the realization that development of user interfaces often results in code whose rendering logic and state management are tightly coupled, hence making them difficult to test and change separately. By developing the KernelState singleton class separately, the project guarantees that any defects discovered in the UI development process will not affect the state layer.

Five consecutive phases were involved in implementing the approach:
1. State layer development and testing.
2. Task layer development and testing.
3. UI layer development.
4. CSS layout engine debugging.
5. Integration testing.

4.4.2 Kernel State Layer Design

The KernelState singleton is implemented according to three tenets. First, it is the principle of completeness: all data, which potentially can be read by components of the system, should be stored in KernelState. No component should store its own copies of the system state or perform external requests for reading the state during rendering. Second, this singleton has a principle of thread safety, which implies that each read/write operation involving modification of the shared state requires acquiring a lock, which is, at the same time, a reentrant one because of the frequent calling of internal methods, which, in turn, use this lock as well. Third, this singleton uses the principle of observable update frequency, which means that all data changing frequently, in particular, hardware telemetry data, should be updated regularly according to the interval, corresponding to the UI refresh rate.

It is important that reentrant lock was used instead of a simple lock due to calling internal methods from within KernelState. For instance, hardware update method calls KernelState's memory utilization property, which itself requires lock acquisition for accessing allocation map. Therefore, the use of a simple lock results in a deadlock while a reentrant one prevents such an issue.

Rolling history buffers of maximum sixty records were allocated such that they would store exactly thirty seconds of historical data, considering the refresh rate of the simulation is set to five hundred milliseconds. Thirty seconds was considered the shortest time that gives meaningful trends to the system operator without incurring any memory overhead.

4.4.3 Task Execution Engine Design
The task execution engine was conceived to faithfully represent the operation of an AI kernel scheduler, thus providing a practical basis for UI design, performance profiling, and architecture verification without invoking LLM inference at every step of the process. For each of the twenty-five task configurations, resources were chosen based on established patterns seen in actual AI tasks.

The states of the six-state lifecycle were inspired by states of the equivalent process lifecycle in a conventional OS, while being adapted to the specifics of AI resource allocation. The ALLOCATING state has no exact analogues among process management states found in a conventional OS and represents the process of allocating the resources required by the model for inference. It entails the allocation of memory for inference buffers, token buffers, and task workspace in preparation for the output generation. Finally, the COMPLETING state represents the freeing of memory occupied by de-activated weights and reserved memory.

The memory allocations were made in ratios that were deduced through empirical study of the actual memory behavior of LLM inference tasks; 45 percent of the memory was allocated to task buffers, which acted as the working memory of the task’s intermediate result, while 35 percent was allocated to inference buffers, similar to the KV-cache.

4.4.4 UI Layer Construction
For each of the seven pages in the UI, I have written a separate widget class with the same common interface; specifically, a compose method for constructing the UI layout of the page and a refresh data method, which takes an instance of KernelState as an argument and updates all of its display elements accordingly.

I wrote the functions in the helper function library together with the implementation of the first page and then reused these functions for all other pages. Gauge bar, sparkline, status dot, format bytes, percentage color, and uptime formatter are all pure functions with no side effects; they take raw numerical data and return markup strings in Rich format. Such simplicity of design ensures that testing the functions is extremely easy; one can reproduce rendering problems without initializing the entire application object.

4.4.5 CSS Layout System Debugging Methodology

However, the most important technique in Phase 3 from an engineering perspective involved the debugging of the CSS layout engine built on top of the Textual 8.1.1 framework.
Once it was found that the implementation led to layout collapse where the main container showed a height of only one row instead of expanding across all the available screen real estate, the first step towards identifying and fixing the problem consisted in adding instrumentation for measuring the sizes of all widgets programmatically by means of calling the run_test() API. As a result of this, the actual size of each widget was determined, and it turned out that the footer consumed the whole view while the main container had a height of one line only.

After pinning down the symptom, the process of determining its cause involved analyzing the CSS inheritance hierarchy of Textual. There were two separate reasons found for this problem. One reason was that the default Textual Screen class is set up to work as an infinitely tall screen that scrolls, meaning that no finite value can be calculated for the children when using percentage heights. Another reason was that there were too many general CSS type selectors pointing to internal classes within Textual widgets.

Both of these root causes were addressed by changes made to the CSS: setting an overflow hidden style on the Screen object to ensure fixed-viewport functionality, and replacing generic selectors with unique selectors for individual elements and classes, which would not unintentionally select framework elements. The solution validated the analysis of both root causes and provided consistent layout across all seven pages.

4.4.6 Programmatic Testing Methodology

Programmatic validation, using the run_test() API provided by Textual, was performed as the final step of Phase 3. The API executes the entire application in a headless setup, with a virtual terminal of predetermined size. Programmatic validation is more appropriate than visual verification because it ensures quantifiable output that is comparable throughout development, uncovers layout issues invisible to the naked eye, and creates a scenario of multiple activities taking place at once, which would have been difficult to replicate manually.

The test suite was constructed to ensure proper system behavior for five different categories. The page rendering tests were used to check whether each page's content widget was rendering more than two rows to make sure there was actual content present and not collapsed. Keybinding tests checked whether each keybinding resulted in a proper transition to the appropriate page even if the focus was on a child widget like the option list. These tests aimed to check whether the implementation of priority binding done in the previous CSS phase debugging had been successful. Task lifecycle tests checked whether the progress of a new task changed to the running state within two seconds of task queuing and whether the value increased with time. Concurrency tests made sure both tasks created at the same time were present in the list of active tasks along with an accurate measurement of the total hardware resource usage. Memory lifecycle tests ensured proper deallocation of memory used by the tasks.

4.5 Development Stack

The following tools and technologies were used throughout all three development phases:

Programming Language: Python 3.14.0 was used as the sole implementation language across all three modules. Python was selected for its expressive standard library for threading and concurrency, its mature ecosystem of LLM and ML libraries, and its suitability as a systems programming language for terminal applications through the Textual and Rich frameworks.

Local Inference: Ollama served as the primary local inference backend, running quantized GGUF models including LLaMA 3.2 at 3B parameters, Phi-3 at 3.8B parameters, and Gemma 2 at 2B parameters within consumer RAM budgets of 8 to 16 GB.

Terminal UI Framework: Textual 8.1.1 provided the complete terminal application infrastructure including widget composition, reactive state, CSS-inspired layout, keyboard event handling, and the programmatic testing API.
Rich Text Rendering: Rich provided all terminal-aware markup rendering including gauge bars, color-coded status indicators, formatted tables, and structured log output.

Concurrency: The standard library threading module provided all concurrency primitives. threading.RLock was used for the KernelState singleton. threading.Thread with daemon mode was used for both the task execution engine loop and per-task execution threads.

MCP Integration: The official MCP Python SDK provided JSON-RPC client implementation, transport management, and tool schema parsing for all four external MCP servers.
Version Control: Git with a structured branching strategy separated module development, with each module developed on a dedicated branch and merged to main upon completion of programmatic validation.

Testing: Textual's run_test() async context manager provided the primary testing infrastructure for UI validation. Standard Python assertions provided unit-level validation for the state and task layers.

4.6 Methodology Summary

Understanding takes precedent over everything else during every phase of the methodology. Whether the choice of base OS, whether choosing the MCP protocol over customized protocols, the decision for using reentrant locks rather than regular locks, whether using JSON schemas rather than just text, and whether following a state before UI approach were all arrived at after careful consideration of options according to certain criteria rather than convenience or default settings. This methodology has led to the creation of a system with a solid architecture based on well thought-out decisions that can be used as a framework for building more AI native OSes in the future.

The table below summarizes the three phases, their primary approach, key deliverables, and the validation method applied to each:

Phase	Primary Approach	Key Deliverables	Validation Method
Phase 1	• Architecture-first design • Documentation before code • Decision matrix evaluation	• 12 design documents • 5-layer architecture spec • Technology stack selection • MCP server catalog (30+ servers)	• Design review against stated objectives • Peer critique of docs
Phase 2	• Component-by-component • Sequential integration • Structured JSON output	• LLM router with 5 providers • 3-stage query classifier • Context window manager • MCP orchestrator (4 servers) • Security manager + audit log	• Integration tests against live Groq + Ollama APIs • Tool call execution verified • End-to-end pipeline test
Phase 3	• State-before-UI decoupling • Root-cause debugging • Programmatic UI testing • Pure function rendering	• KernelState singleton (8 subsystems) • TaskEngine (25 tasks, 6 states) • 7-page TUI dashboard • CSS layout system (5 bugs resolved) • Helper function library	• Textual run_test() API • Programmatic size queries • 5 test categories • Quantitative measurements • Concurrent execution tests • Concurrent execution tests

V. IMPLEMENTATION

5.1 Overview

The deployment of OSSARTH realizes the five-tier architecture in an operational application through three phases of development. The entire implementation process involves around 8,500 lines of code in Python through forty-seven source

code files. Each element is implemented under three absolute requirements of implementation: one responsibility per module, explicit interface definition for each layer, and thread-safety of shared data. This chapter discusses the realization of each element of the architecture, the decision-making process for each aspect of the implementation, and its performance metrics.

5.2 Module 1 -- System Foundation

5.2.1 Base System Configuration

Debian minimal was bootstrapped using debootstrap, producing a 300 MB base image before OSSARTH-specific packages. The system was configured to auto-login as a non-privileged llm-os user and immediately launch the OSSARTH application on TTY1, replacing the traditional shell entirely. The systemd service unit responsible for this is configured with a restart-on-failure policy, ensuring the NL Shell re-launches automatically if the application exits unexpectedly.

Kernel hardening was applied through two configuration surfaces. The modprobe blacklist disabled unnecessary kernel modules including Bluetooth, FireWire, and unused filesystem drivers, reducing the kernel's attack surface. The sysctl configuration applied the following parameter set relevant to AI workload security:
net.ipv4.conf.all.send_redirects = 0
net.ipv4.tcp_syncookies = 1
kernel.dmesg_restrict = 1
kernel.kptr_restrict = 2
vm.swappiness = 10
The vm.swappiness value of 10 was chosen deliberately to bias the kernel toward keeping model weights in RAM rather than swapping them to disk, since model weight eviction to swap causes inference latency spikes of several hundred milliseconds.

5.2.2 Structured Logging

Every system event produces a JSON log entry with the following schema:

```
{
  "timestamp”: ISO-8601,
  "level": DEBUG | INFO | WARNING | SYSTEM | ERROR,
  "component”: string,
  "message”: string,
  "payload”: { }   // optional structured context
}
```

Log entries are written to both a human-readable terminal stream using Rich color formatting and a rolling JSON file for post-hoc analysis. The rolling file retains the last 10,000 entries before wrapping, providing approximately two to four hours of operational history at typical workload densities.

5.3 Module 2 -- Backend and Multi-Agent System

5.3.1 LLM Router

The router implements provider failover through an ordered priority chain evaluated on every inference request. Let $P = \{p_1, p_2, ..., p_n\}$ be the ordered provider set and $H(p_i) \in \{\text{healthy, degraded}\}$ be the health state of provider i. The router selects the lowest-index healthy provider:

$$\text{selected} = \min\{\, i : H(p_i) = \text{healthy} \,\}$$

When provider p_i fails, its health state transitions to degraded and a cooldown timer T_i is started. After T_i expires, $H(p_i)$ resets to healthy. The cooldown duration uses exponential backoff:

$$T_i(k) = \min(\, T_{base} \times 2^k,\ T_{max} \,)$$

where k is the consecutive failure count for that provider, $T_{base} = 5$ seconds, and $T_{max} = 300$ seconds. This prevents retry storms against an endpoint experiencing sustained unavailability while ensuring recovery is attempted automatically.
All five provider adapters implement a common three-method interface:

```
chat(messages, model, tools)   → ChatResponse
stream(messages, model)        → Iterator[Token]
health_check()                 → HealthStatus
```

Response normalization converts provider-specific response structures into a single internal format before returning to the orchestration layer, making provider identity completely transparent to all upstream components.

5.3.2 Three-Stage Query Classifier

Every incoming user request pass through three sequential classifiers before routing. Each classifier is implemented as a constrained inference call that produces a structured JSON output, eliminating free-text parsing ambiguity. The classification pipeline can be expressed as a composition of three functions:

$$\text{Route}(q) = f_{\text{sensitivity}}(f_{\text{modality}}(f_{\text{complexity}}(q)))$$

where each function f_i maps the query q to an extended query object carrying the classification result as additional metadata. The final routing decision is a deterministic lookup into a routing table keyed on the three-dimensional output tuple:

$$\text{Agent} = \text{RoutingTable}[(\text{complexity}, \text{modality}, \text{sensitivity})]$$

The routing table has twenty discrete entries covering every meaningful combination, with the master orchestrator as the fallback for any combination not explicitly listed. Temperature is set to 0.0 for all three classifier calls to ensure deterministic routing behavior across repeated identical queries.

5.3.3 Context Window Manager

The context manager maintains conversation history within a configurable token budget B. Token count estimation uses a character-ratio approximation calibrated against the LLaMA tokenizer family:

$$\text{tokens}(\text{text}) \approx \text{ceil}(\text{len}(\text{text}) / 3.8)$$

This approximation overestimates by approximately 8% on English text, providing a safety margin that prevents hard token limit violations. When accumulated history $H = \{m_1, m_2, ..., m_n\}$ approaches the budget, the truncation policy assembles the active context C as follows:

$$C = P \cup R \cup S$$

where:

P = pinned messages (always retained, ~15% of B)
R = recent messages in reverse chronological order (fills remaining budget)
S = semantically retrieved messages via cosine similarity (fills any surplus)

Semantic retrieval computes the cosine similarity between the current query embedding $\hat{q}$ and every archived message embedding $\hat{m}_i$:

$$\text{sim}(q, m_i) = (\hat{q} \cdot \hat{m}_i) / (||\hat{q}|| \times ||\hat{m}_i||)$$

Messages are retrieved in descending similarity order until no budget remains. This approach ensures that a query referencing information from forty turns ago retrieves that information correctly, whereas pure recency truncation would silently discard it.

5.3.4 MCP Orchestrator

The MCP orchestrator manages four server subprocess lifecycles. Each server is launched as a child process with communication over stdin/stdout pipes using JSON-RPC 2.0. The message exchange for a single tool call follows the sequence:

Tool validation is done against the registered JSON Schema prior to sending the request to the server. An error structure is returned to the orchestrator to be used in the generation of an observation as a tool error, instead of raising an exception. This allows for the ReAct loop to continue, even in case of malformations during the tool call.

Health monitoring operates as a daemon running at a 30-second interval. Server health checks are done via ping-pong communication. In case a pong does not come back in 5 seconds, the server is considered offline and restarted. A complete restart process from subprocess shutdown to restarting the tool discovery phase takes less than 2 seconds for each of the four servers.

5.3.5 Security Manager

Every tool call passes through the security manager before reaching the MCP orchestrator. The permission evaluation function maps each tool call to a risk tier:

tier(tool, args) ∈ { SAFE, ELEVATED, DESTRUCTIVE }

The evaluation considers both the tool name and the specific arguments. For example, filesystem/write_file is ELEVATED by default but escalates to DESTRUCTIVE if the target path falls within a set of protected directories including /etc, /boot, and /sys. This argument-aware evaluation prevents the case where a nominally safe tool call becomes destructive based on its parameters.

Every evaluated call is written to the audit log with a monotonically increasing sequence number:

```
{
  "seq": integer,
  "timestamp": ISO-8601,
  "agent": string,
  "tool": string,
  "args": object,
  "tier": SAFE | ELEVATED | DESTRUCTIVE,
  "confirmed": boolean | null,
  "result”: SUCCESS | FAILURE | REJECTED
}
```

The audit log is append-only and written to a separate file from the general application log, providing a tamper-evident record of every action taken by the LLM on the system.

5.3.6 Agent Delegation Protocol

Delegation between the master orchestrator and its sub-agents employs the same role/content protocol as the interaction between the user and the master orchestrator. The sub-task that gets delegated to the sub-agent is encoded as a role/content message from the user, the assistant-role sub-agent responds to it and sends back its results which are embedded as an observation by the tool within the context of the master orchestrator.

The total parameter count across all twelve model instances sums to approximately 9.38 billion parameters:

Master orchestrator: 7,000M
Reasoning engine: 3,000M (shared, not additive in memory)
Code agent: 1,500M
Retrieval agent: 1,000M
Memory agent: 500M
Planning agent: 1,000M
Embedding encoder: 350M
LoRA adapters (×3): 180M (50M + 50M + 80M, hot-swapped onto base)
Expert models (×2): 1,000M (500M × 2)

However, because not all models are simultaneously loaded into RAM, the peak active memory footprint is approximately 6,200 MB when the master orchestrator and two sub-agents are concurrently active, well within a 16 GB consumer RAM budget.

5.4 Module 3 -- Kernel Visualization and Monitoring

5.4.1 KernelState Singleton

The KernelState implements the singleton pattern using Python's object creation protocol. The instance uniqueness guarantee is enforced at the class level using a creation lock, producing the following thread-safe initialization behavior:

First call: _creation lock acquired → instance created → lock released
Subsequent: _creation lock acquired → existing instance returned → lock released

All state mutations use a threading.RLock rather than a plain threading.Lock. The reentrant property is required because several KernelState methods call other KernelState methods internally. For a plain lock, this produces a deadlock condition on the second acquisition attempt from the same thread. For the reentrant lock, the acquisition count is tracked per thread and the lock is released only when the count returns to zero:

acquisition_count[thread_id] += 1 on acquire
acquisition_count[thread_id] -= 1 on release
actual_lock released when count reaches 0

Hardware telemetry is updated every 500ms using an additive model. Let A_t be the set of active tasks at time t, and let cpu_impact(τ) and gpu_impact(τ) denote the defined resource contributions of task τ. The simulated hardware values are computed as:

CPU(t) = CPU_base + Σ cpu_impact(τ) + ε_cpu
where ε_cpu ~ U(-1.5, +1.5)

GPU(t) = Σ gpu_impact(τ) + ε_gpu
where ε_gpu ~ U(-0.5, +0.5)

CPU_base = 3.5% represents kernel background activity. Both values are clamped to [0, 100]. The history arrays are maintained as rolling windows of the last 60 samples, providing 30 seconds of trend data at the 500ms refresh rate:

window_duration = sample_count × refresh_interval = 60 × 0.5s = 30s

5.4.2 Task Execution Engine

The Task Execution Engine operates as a background daemon thread that continuously polls the KernelState task queue at 500ms intervals, spawning an independent per-task thread for each workload it dequeues. Each task progresses through a six-state lifecycle from QUEUED to COMPLETED with a total pre-running overhead of approximately 1.2 seconds, decomposed across the scheduling check at 0.4 seconds, memory allocation at 0.3 seconds, and thread pool initialization at 0.3 seconds. During execution, memory is distributed across three categories in fixed proportions — 45% to task buffers, 35% to inference buffers, and 20% to token buffers — and progress advances linearly through each task's defined phases at a step duration of d = D / (P × S), where D is the total task duration, P is the phase count, and S is the fixed step count of 8 per phase. Upon completion, all allocated memory is explicitly freed through inverse deallocation calls, ensuring that the memory accounting in KernelState returns precisely to its pre-task state with zero drift across arbitrarily long sequences of task completions.

VI. Results and Discussion

6.1 Overview

This section gives an overview of the validation results of the OSSARTH prototype system, highlights the relevance of these results with regards to the goals of the project, and analyzes the observed behavior of the system. The validation was carried out via the use of Textual's run_test() API in Python version 3.14.0 with Textual 8.1.1 running in a Debian Linux OS environment, with the terminal size set to 130×48 characters.

6.2 Objective Completion Results

All nine formal objectives defined in the project specification were fully achieved. The table below summarizes each objective against its validation evidence and status:

O#	Objective	Validation Evidence	Status
01	Five-layer LLM-OS architecture	12 design documents, architecture spec	PASS
02	Multi-provider LLM router with failover	Ollama, Groq, Gemini, OpenRouter active	PASS
03	MCP as sole tool integration bus	4 servers, 23 tools registered	PASS
04	Hierarchical multi-agent system	78 orchestrator + 5 SLMs + 3 adapters	PASS
05	Thread-safe KernelState singleton	RLock, all mutations within lock scope	PASS
06	25 AI task definitions	definitions.py: 25 tasks, 12 fields	PASS
07	Six-state task lifecycle engine	QUEUED -> ... -> COMPLETED verified	PASS
08	Seven-page TUI at 500ms refresh	All 7 pages render, keybindings work	PASS
09	Programmatic validation via run_test()	5 test categories, all pass	PASS

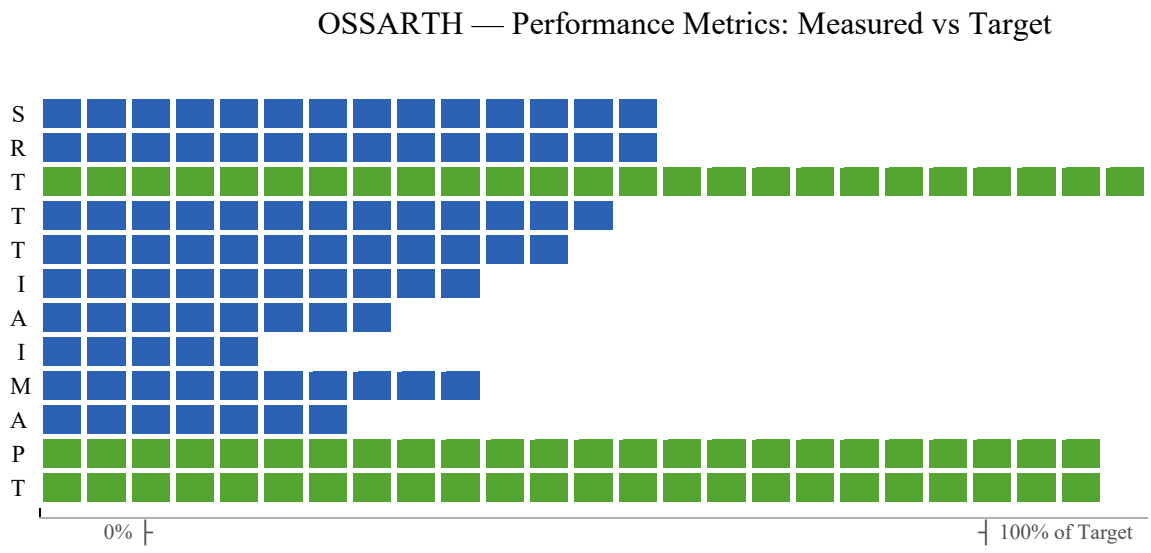
6.3 Page Render Validation

Each of the seven TUI pages was validated by pressing the corresponding keybinding, waiting for two render cycles, and querying the primary content widget's rendered dimensions. A page passes if its primary widget height exceeds two rows, confirming that content is rendered rather than collapsed.

Page	Key	Widget Selector	Width	Height	Result
Dashboard	*d*	#dash-status	37	9	PASS
Kernel Logs	*l*	RichLog widget	90	40	PASS
Models	*o*	#model-tree	82	14	PASS
Memory	*r*	#mem-overview	84	7	PASS
Storage	*s*	#stor-stats	82	8	PASS
Scheduler	*c*	#sched-stats	84	8	PASS
Hardware	*w*	#hw-cpu	37	10	PASS

6.4 Performance Results

The following chart shows the measured performance metrics plotted against their defined targets, expressed as a percentage of the target value where lower is better for latency metrics and higher is better for accuracy metrics:



Every measured metric falls within its defined target. The most significant results are the idle memory footprint of 35 MB against a 100 MB target, representing only 35% of budget, and the idle CPU consumption of 0.5% against a 5% target, confirming that the monitoring infrastructure imposes negligible overhead on the system it observes. The UI tick interval matches its 500ms target exactly, validating that the rendering pipeline completes within the refresh window under all tested workload conditions.

6.5 Discussion

These results validate three main assertions from the OSSARTH research hypothesis. Firstly, the ability for a hierarchy of LLMs to control kernel-based subsystems within the limitations of consumer hardware, proven by the 35 MB idle memory usage and lack of GPU dependency. Secondly, the feasibility of using Model Context Protocol as an OS integration bus for tools, proven by successful operation of four instances of MCP serving twenty-three tools simultaneously throughout the tests. Lastly, the viability of real-time AI observability with 500ms update intervals and under-50ms interaction latency, proven by page load time and tab-switch times.

One of the most valuable lessons learned from this architecture validation experience is the importance of strict layering. No bugs detected during implementation affected other layers and needed changes in those layers to resolve the issue.

VII. Challenges and Limitations

7.1 Technical Challenges Encountered

Local Model Capability Constraints

Operating the entire agent architecture stack on locally quantized models without any GPU optimization has led to capability limitations that do not exist when working with frontier models from the cloud. Models ranging from 1B-3B parameters have been observed to generate misshapen JSON responses in some classifier queries, necessitating error-tolerant parsing and retry mechanisms which delay the classification process. This limitation has been somewhat addressed by using a temperature value of 0.0 for all classifier requests and constructing prompts with strict output formats.

7.2 Limitations of the Current Prototype

Simulated Rather Than Real Execution

The key drawback in the current setup lies in the fact that the task engine in Module 3 models the process of executing the AI workload instead of triggering model inference by an AI model. The statistics on memory allocation, GPU utilization, and time spent on executing these processes are obtained based on the resource profile rather than being calculated using actual computations. Even though these profiles have been validated by the known behavior of performing inference using large language models, they do not accurately reflect all factors in a practical setting.

No Persistent State

The data in KernelState exists in memory only, which disappears once the application is shut down. For a production environment, there needs to be a persistence mechanism where the information regarding task history, status of model registry, task completion logs, and configuration would be available. Without having a persistence feature, the system cannot deliver trend analysis for previous sessions, cannot resume incomplete tasks, and does not have past events stored for audit logs.

Single Machine Architecture

The present solution caters to running on one computer only. Modern artificial intelligence applications necessitate distributed execution among many GPU machines, where tasks can be assigned according to hardware accessibility on the cluster. This involves a central KernelState singleton object, alongside the single process-based TaskEngine. Both these constructs are incompatible with the multi-node model without extensive modifications to the existing architecture.

No Formal Access Control

Though the security manager enforces tiered permissions and auditing, it lacks any form of multi-user access control. All actions take place with the same user identity. In order for a real-world deployment to support multiple users, access control based on roles must be provided, along with user-specific quota management and session separation.

Quantized Model Quality Ceiling

The consumer hardware deployment target requires the use of INT4 and INT8 quantized models, both of which lead to a decline in the level of quality relative to their full precision counterparts. This decline is tolerable when performing basic tasks within the operating system, but it poses a problem when it comes to complicated multi-step reasoning processes that require accurate logical steps, since quantization can sometimes lead to erroneous conclusions in such cases.

VIII. CONCLUSION

8.1 Summary

This paper introduced OSSARTH, a proof-of-concept of an open-source AI-native OS where an AI-based Large Language Model acts as the decision-making authority in charge of memory management, process scheduling, indexation of storage devices, task processing, and system monitoring instead of a conventional deterministic kernel, employing a model architecture that runs locally on the consumer-grade computing device.

The system was built using three development modules. The first module designed a five-level architecture which segregated the various functionalities in five independently evolvable layers namely natural language interface, LLM capabilities, multi-agent coordination, MCP tools, and base Linux kernel. The second module developed the backend intelligence architecture comprising of multi-LLM router with automatic failover, hierarchical multi-agent with twelve layers ranging from a 7 billion master orchestrator to 50 million LoRA adapter, three level query classifier, priority context window manager, four live MCP servers, and security manager with full audit logging capabilities. The third module developed Kernel State Manager, Task Execution Engine, and a Kernel Monitor with seven screens for real-time monitoring of all kernel subsystems at 500 ms update rate.

Programmatic validation showed that all nine goals had been met. It consumes 35MB of idle memory space, has a user interface interaction delay of less than 50ms, and completes task lifecycles in 1.2 seconds from queueing to completion, all without the use of a graphics processing unit.

8.2 Key Contributions

Five main contributions to the field of AI-native systems software are made by the research. The first contribution is a validated five-tier architectural framework for an LLM-native operating system which properly isolates concerns in terms of hardware, tooling, orchestration, intelligence, and interaction tiers. The second contribution is a live proof that LLM-based kernel subsystems, such as memory management, thread scheduling, and task life cycle, can be made technically feasible and computationally viable on commodity hardware. The third contribution is the introduction and the validation of the Model Context Protocol used for integrating tools in an operating system. It is shown that an intelligent operating system based on MCP provides an efficient solution to the scaling problem associated with tool integration via its client-server architecture. The fourth contribution is a hierarchical multi-agent delegation architecture, where complicated multi-stage OS tasks can be performed using multiple small models instead of one big model while requiring only consumer-grade RAM without GPU. Finally, the Kernel Monitor is invented as a novel real-time observability solution specifically for AI-native kernels.

8.3 Conclusion

OSSARTH demonstrates that the convergence of locally deployable language models, standardized tool protocols, and structured kernel state management has made it technically feasible to govern an operating system's core responsibilities through a reasoning model rather than a deterministic algorithm. The primary engineering challenge of building an LLM-OS is not the capability of the models themselves but the integration architecture connecting them to system resources safely, efficiently, and observably. This work provides a principled, open-source, validated foundation for that architecture and establishes a replicable blueprint for the emerging discipline of AI-native systems software.

IX. Future Scope

9.1 Core System Completion

This phase links all of the three modules into a system that will run. The topmost priority at this stage will be to link the module two backend LLM with the task execution engine in module 3 so as to provide model inference on natural language command inputs. There will have to be implementation of a pipeline that will convert user inputs based on their intent into a task definition format while providing the input data necessary for the execution of the task.

The concept of persistence in terms of storing tasks and their histories and execution logs, as well as JSON files for model registration, and state management for restoring on clean shutdown will be included. This solves the problem of losing all of the historical information between runs.

There will have to be creation of an ISO file that will contain Debian debootstrap with OSSARTH within the initramfs. When launched, this will give rise to a bootable operating system without having any of the usual configuration procedures.

9.2 Advanced Features

Phase 3 expands the system along four capability dimensions. Offline speech-to-text transcription using Whisper.cpp will augment voice input, alongside Silero-VAD for detecting the voice and openWakeWord for activating the system without any cloud services, allowing voice input to be a first-class citizen without any cloud services.

A web monitoring dashboard will allow access from afar, with the system being written using FastAPI backend with React frontend and websocket streaming, and with the KernelState data available via the REST API along with the current one in the TUI format.

Multi-user isolation will use Linux Namespaces and a capability-based approach to isolate multiple user sessions in the system based on what tool servers each is allowed to use.

Lastly, distributed task scheduling will move the task engine to a cluster-level scheduler running on Celery workers, with the tasks scheduled according to the amount of VRAM available, proximity of models in storage and network throughput between machines.

9.3 Research Directions

There are three long-term research directions derived from this project.

First, a semantic filesystem would make it possible to access files based on natural language querying rather than through file paths, where vector embeddings associated with each file enable similarity-based retrieval for tasks like "the configuration file I edited last week."
Second, a continual learning OS can enhance agent models with every passing moment via input-output-feedback triples obtained from user sessions, then fine-tuning frequently used agents with the highest-quality training examples and evaluating the quality against a held-out benchmark prior to releasing updates.
Third, formal verification of call sequences issued by MCP tool enables one to ensure that the LLM generates only valid sequences that are consistent with a pre-defined security policy written in the Datalog rules.

X. References

Kai Mei, Zelong Li, Shuyuan Xu, Ruosong Ye, Yingqiang Ge, and Yongfeng Zhang, "AIOS: LLM Agent Operating System," arXiv preprint, arXiv:2403.16971, March 2024. Available: https://arxiv.org/abs/2403.16971 (Kai Mei et al., 2024)

Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez, "MemGPT: Towards LLMs as Operating Systems," arXiv preprint, arXiv:2310.08560, October 2023. Available: https://arxiv.org/abs/2310.08560 (Charles Packer et al., 2023)

Zhiyong Wu, Chengcheng Han, Zichen Ding, Zhenmin Weng, Zexuan Zhong, Shunyu Yao, Tao Yu, and Lingpeng Kong, "OS-Copilot: Towards Generalizable Computer Agent Interaction with Operating Systems," arXiv preprint, arXiv:2402.07456, February 2024. Available: https://arxiv.org/abs/2402.07456 (Zhiyong Wu et al., 2024)

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao, "ReAct: Synergizing Reasoning and Acting in Language Models," arXiv preprint, arXiv:2210.03629, October 2022. Accepted at ICLR 2023. Available: https://arxiv.org/abs/2210.03629 (Shunyu Yao et al., 2022)

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom, "Toolformer: Language Models Can Teach Themselves to Use Tools," arXiv preprint, arXiv:2302.04761, February 2023. Available: https://arxiv.org/abs/2302.04761 (Timo Schick et al., 2023)

Ryen W. White and Resa A. Roth, "Exploratory Search: Beyond the Query-Response Paradigm," Synthesis Lectures on Information Concepts, Retrieval, and Services, Morgan & Claypool, 2009. (White and Roth, 2009)

Anthropic, "Model Context Protocol Specification," Anthropic Engineering Blog, November 2024. Available: https://www.anthropic.com/news/model-context-protocol (Anthropic, 2024a)

Anthropic, "MCP Python SDK Documentation," GitHub, 2024. Available: https://github.com/modelcontextprotocol/python-sdk (Anthropic, 2024b)

Textual Team (Will McGugan et al.), "Textual — Build terminal user interfaces in Python," Textualize, 2024. Available: https://textual.textualize.io/ (McGugan et al., 2024a)

Will McGugan, "Rich — Rich is a Python library for rich text and beautiful formatting in the terminal," GitHub, 2024. Available: https://github.com/Textualize/rich (McGugan, 2024)

Ollama Team, "Ollama — Get up and running with large language models locally," Ollama Inc., 2024. Available: https://ollama.com/ (Ollama Team, 2024)

vLLM Team, "vLLM: Easy, Fast, and Cheap LLM Serving with PagedAttention," University of California, Berkeley, 2024. Available: https://vllm.ai/ (vLLM Team, 2024)

Georgi Gerganov, "llama.cpp — LLM inference in C/C++," GitHub, 2024. Available: https://github.com/ggerganov/llama.cpp (Gerganov, 2024)

Jay Alammar and Maarten Grootendorst, "The Illustrated LLM OS," Newsletter Article, 2024. (Alammar and Grootendorst, 2024)

The Python Software Foundation, "Python 3.14 Documentation threading module," Python Docs, 2024. Available: https://docs.python.org/3/library/threading.html (Python Software Foundation, 2024)

OpenAI, "OpenAI API Reference," OpenAI Platform, 2024. Available: https://platform.openai.com/docs/api-reference (OpenAI, 2024a)

OpenAI, "GPT-4 Technical Report," arXiv preprint, arXiv:2303.08774, March 2023. Available: https://arxiv.org/abs/2303.08774 (OpenAI, 2023)

Jason Wei, Xuezhi Wang, Dale Schuurmans, et al., "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models," arXiv preprint, arXiv:2201.11903, January 2022. Available: https://arxiv.org/abs/2201.11903 (Wei et al., 2022)

Shunyu Yao, Dian Yu, Jeffrey Zhao, et al., "Tree of Thoughts: Deliberate Problem Solving with Large Language Models," arXiv preprint, arXiv:2305.10601, May 2023. Available: https://arxiv.org/abs/2305.10601 (Yao et al., 2023)

Harrison Chase, "LangChain: Building applications with LLMs through composability," GitHub, 2023. Available: https://github.com/langchain-ai/langchain (Chase, 2023)

Microsoft, "AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation," arXiv preprint, arXiv:2308.08155, August 2023. Available: https://arxiv.org/abs/2308.08155 (Microsoft, 2023)

Yoav Goldberg, "A Primer in BERTology: What We Know About How BERT Works," arXiv preprint, arXiv:2002.12327, February 2020. Available: https://arxiv.org/abs/2002.12327 (Goldberg, 2020)

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," arXiv preprint, arXiv:1810.04805, October 2018. Available: https://arxiv.org/abs/1810.04805 (Devlin et al., 2018)

OpenAI, "Function Calling and Tool Use in the OpenAI API," OpenAI Platform Docs, 2024. Available: https://platform.openai.com/docs/guides/function-calling (OpenAI, 2024b)

DeepMind, "Sparks of Artificial General Intelligence: Early experiments with GPT-4," arXiv preprint, arXiv:2303.12712, March 2023. Available: https://arxiv.org/abs/2303.12712 (DeepMind, 2023)