UNIVERSITY OF PIRAEUS - DEPARTMENT OF INFORMATICS

MSc "Cybersecurity and Data Science"
ΠΜΣ "Κυβερνοασφάλεια και Επιστήμη Δεδομένων"

MSc Thesis

Thesis Title: Τίτλος Διατριβής:	**Specialized Multi-Agent Autonomous Coordination for Complex Project Execution using Balanced Cooperation, Dynamic Virtualized Playgrounds, and Extensible Tool Sets** Αυτόματος Συντονισμός Πολλαπλών Εξειδικευμένων Πρακτόρων για την Εκτέλεση Πολύπλοκων Έργων με χρήση Ισορροπημένης Συνεργασίας, Δυναμικών Εικονικών Μηχανών και Επεκτάσιμων Συνόλων Εργαλείων
Student's name-surname: Ονοματεπώνυμο φοιτητή:	**Spyridon Fotopoulos** Σπυρίδων Φωτόπουλος
Father's name: Πατρώνυμο:	**Vasileios** Βασίλειος
Student's ID No: Αριθμός Μητρώου:	ΜΠΚΕΔ24045
Supervisor: Επιβλέπων:	**Dionysios Sotiropoulos, Assistant Professor** Διονύσιος Σωτηρόπουλος, Αναπληρωτής Καθηγητής

April 2026 / Απρίλιος 2026

3-Member Examination Committee

Dionisios Sotiropoulos	Efthimios Alepis	Athanassios Avgerinos
Associate Professor	Professor	Assistant Professor

Copyright

© 2026 Fotopoulos Spyridon. All rights reserved.

This work is submitted exclusively for the purpose of academic evaluation and fulfillment of degree requirements. No part of this thesis may be reproduced, stored, or transmitted in any form or by any means - digital, physical, or otherwise - and for any other reason, without the explicit prior written permission of the author. The views and conclusions expressed in this document are those of the author and do not represent the official positions of the University of Piraeus. As the author of this paper, I declare that this paper does not constitute a product of plagiarism and does not contain material from unquoted sources.

Abstract

The rapid advancement of Large Language Models (LLMs) has, at the same time, initiated a shift from simple conversational interfaces to autonomous Multi-Agent Systems (MAS) capable of executing complex technical workflows. However, current orchestration frameworks face three critical bottlenecks: the "memory wall" where context degradation leads to hallucinations in long-horizon tasks; vendor lock-in through tight coupling with proprietary APIs; and significant security risks associated with running agent-generated code without hardware-level isolation.

This thesis introduces **SandBot** (an Agent Sandboxing & Orchestration Framework), an enterprise-grade MAS architecture designed to bridge the gap between experimental research and secure production deployment. SandBot addresses context exhaustion through a "Six-Section Prompt Architecture" and an algorithmic "Tool Description Deduplication" technique that reduces Model Context Protocol (MCP) overhead by approximately 47%. To ensure operational safety, the framework programmatically integrates with **OpenNebula** to provision temporary, network-isolated MicroVMs for agentic tasks, enforcing isolation that mitigates Arbitrary Code Execution (ACE) risks.

Central to the framework is an **Orchestrator-Delegate pattern** characterized by balanced peer-to-peer cooperation. This topology replaces rigid hierarchies with a flat network of specialized peers - standardized as "Atomic Agents" - that interact via isolated namespaces. By decoupling orchestration logic from the underlying cognitive models (Bring-Your-Own-Model), SandBot provides a sovereign, future-proof, and secure path for the autonomous execution of complex technical projects.

Subject area: Artificial Intelligence, Distributed Systems, and Cybersecurity.

Keywords: Multi-Agent Systems, LLM Orchestration, Model Context Protocol, Virtualized Sandboxing, OpenNebula, Token Optimization, Autonomous AI.

Περίληψη

Η ραγδαία εξέλιξη των Μεγάλων Γλωσσικών Μοντέλων (Large Language Models - LLMs) πυροδότησε ταυτόχρονα τη μετάβαση από απλές διεπαφές συνομιλίας σε αυτόνομα Πολυπρακτορικά Συστήματα (Multi-Agent Systems - MAS), ικανά να εκτελούν πολύπλοκες τεχνικές ροές εργασίας. Ωστόσο, τα υφιστάμενα frameworks ενορχήστρωσης (orchestration frameworks) αντιμετωπίζουν τρία κρίσιμα εμπόδια: το «τείχος μνήμης» (memory wall), όπου η υποβάθμιση του context (context degradation) οδηγεί σε ψευδαισθήσεις (hallucinations) σε μακροσκελείς εργασίες, τον εγκλωβισμό σε συγκεκριμένους παρόχους (vendor lock-in) λόγω στενής δέσμευσης με συγκεκριμένα API, και τους σημαντικούς κινδύνους ασφαλείας που σχετίζονται με την εκτέλεση κώδικα παραγόμενου από πράκτορες χωρίς απομόνωση σε επίπεδο υλικού (hardware-level isolation).

Η παρούσα διατριβή παρουσιάζει το SandBot (ένα Πλαίσιο Απομόνωσης & Ενορχήστρωσης Πρακτόρων - Agent Sandboxing & Orchestration Framework), μια αρχιτεκτονική MAS επαγγελματικού επιπέδου, σχεδιασμένη να γεφυρώσει το χάσμα μεταξύ της πειραματικής έρευνας και της ασφαλούς ανάπτυξης σε περιβάλλον παραγωγής. Το SandBot αντιμετωπίζει την εξάντληση του context μέσω μιας «Αρχιτεκτονικής Προτροπών Έξι Σημείων» (Six-Section Prompt Architecture) και μιας αλγοριθμικής τεχνικής Συμπίκνωσης Περιγραφής Εργαλείων» (Tool Description Deduplication), η οποία μειώνει τον επιπλέον φόρτο (overhead) του Πρωτοκόλλου Πλαισίου Μοντέλου (Model Context Protocol - MCP) κατά περίπου 47%. Για τη διασφάλιση της λειτουργικής ασφάλειας, το πλαίσιο ενσωματώνεται προγραμματιστικά με το OpenNebula για την παροχή προσωρινών, δικτυακά απομονωμένων Μικρο-Εικονικών Μηχανών (MicroVMs) για τις εργασίες των πρακτόρων, επιβάλλοντας αυστηρή απομόνωση που μετριάζει τον κινδύνο Αυθαίρετης Εκτέλεσης Κώδικα (Arbitrary Code Execution - ACE).

Κεντρικό στοιχείο του framework αποτελεί ένα μοτίβο Ενορχηστρωτή-Αντιπροσώπου (Orchestrator-Delegate pattern), το οποίο χαρακτηρίζεται από ισορροπημένη ομότιμη συνεργασία (peer-to-peer cooperation). Αυτή η τοπολογία αντικαθιστά τις στατικές ιεραρχίες με ένα επίπεδο δίκτυο εξειδικευμένων ομότιμων κόμβων - τυποποιημένων ως «Ατομικοί Πράκτορες» (Atomic Agents) - που αλληλεπιδρούν μέσω απομονωμένων χώρων ονομάτων (isolated namespaces). Αποσυνδέοντας τη λογική ενορχήστρωσης από τα υποκείμενα γνωστικά μοντέλα (προσέγγιση Bring-Your-Own-Model), το σύστημα SandBot παρέχει μια ανεξάρτητη (sovereign), διαχρονική και ασφαλή κατεύθυνση για την αυτόνομη εκτέλεση πολύπλοκων τεχνικών έργων.

Γνωστικό Αντικείμενο (Θεματική Περιοχή): Τεχνητή Νοημοσύνη, Κατανεμημένα Συστήματα και Κυβερνοασφάλεια.

Λέξεις-κλειδιά: Πολυπρακτορικά Συστήματα, Ενορχήστρωση LLM, Πρωτόκολλο Πλαισίου Μοντέλου, Εικονικοποιημένη Απομόνωση (Sandboxing), OpenNebula, Βελτιστοποίηση Διακριτικών (Token Optimization),

Αυτόνομη Τεχνητή Νοημοσύνη.

Acknowledgements

I would like to express my gratitude to my supervising professor, Dr. Dionisios Sotiropoulos. His guidance has been invaluable throughout this journey.

I am also sincerely grateful to Dr. Thanassis Avgerinos, Assistant Professor at the National and Kapodistrian University of Athens, for his time and expertise. His suggestions and the perspectives shared during our discussions greatly enriched the quality of this work.

I am particularly appreciative of my friend and former classmate, Nikos Dalkos, for our productive collaboration. The exchange of ideas and the shared commitment to our objectives greatly enhanced the research process; it has been a privilege to work alongside such a dedicated peer.

Furthermore, I would like to thank my family and friends for their support and for providing a necessary balance during the most demanding periods of my studies.

Table of Contents

Copyright........ 2
Abstract........ 3
Περίληψη........ 4
Acknowledgements........ 5
Table of Contents........ 6
List of Figures........ 9
List of Images........ 9
List of Abbreviations........ 9
1. INTRODUCTION........ 10
1.1 Motivation........ 10
1.2 Problem Statement........ 12
1.3 Research Questions........ 13
1.4 Objectives and Scope........ 14
1.5 Summary of Contributions........ 16
1.6 Thesis Structure........ 17
2. BACKGROUND AND RELATED WORK........ 18
2.1 Multi-Agent Systems (MAS) and Stateful Orchestration........ 19
2.1.1 The Convergence of Reasoning and Action (ReAct)........ 19
2.1.2 From Rigid Hierarchies to Graph-Centric State Machines........ 19
2.1.3 Topologies and the Science of Scaling........ 20
2.2 Extensible Tooling and the Model Context Protocol (MCP)........ 21
2.3 Dynamic Virtualized Playgrounds and Security........ 22
2.4 Context Optimization and Stateful Memory Architectures........ 23
2.4.1 The Cognitive Taxonomy of LLM Memory........ 23
2.4.2 The Mathematical and Empirical "Memory Wall"........ 23
2.4.3 External Memory: Vector RAG versus GraphRAG........ 24
2.5.1 The Dimensions of Vendor Lock-in and the "Unreliability Tax"........ 25
2.5.2 Economic Routing and Hybrid Execution (BYOM)........ 25
2.6 Comparative Analysis of Current Orchestration Frameworks........ 26
3. OVERVIEW OF SANDBOT (MACRO-ARCHITECTURE)........ 27
3.1 Conceptual Model: The Orchestrator-Delegate Pattern........ 28
3.2 The Three-Layer System Architecture........ 30
3.2.1 Frontend Layer (Observability and Presentation)........ 30
3.2.2 Backend Layer (API Gateway and Session Management)........ 31
3.2.3 Framework Layer (Stateful Cognitive Engine)........ 31
3.3 Infrastructure Resilience and Graceful Degradation........ 33
3.3.1 The Three-Layer Provisioning........ 33
3.4 Middleware Pipeline and Extensibility........ 35
3.5 Security-First Design Principles and Memory Isolation........ 36
3.5.1 Namespace Isolation and Sub-Agent Lifecycles........ 36
4. SANDBOT METHODOLOGY: ORCHESTRATION, MEMORY, AND OPTIMIZATION........ 37
4.1 Peer-to-Peer Coordination and the User-Facing Orchestrator........ 38

4.1.1 Sub-Agent Spawning and Namespace Isolation........ 38
4.2 The Atomic Agent: A Four-Component Anatomy........ 39
4.2.1 Agent Immutability versus Customization........ 39
4.3 Context Construction and the Six-Section Prompt........ 40
4.3.1 Operational Strategies: Efficiency and Pre-Completion Checks........ 40
4.4 Tool Abstraction and Context Optimization........ 41
4.5 Knowledge Management: Active vs. Passive RAG........ 43
4.5.1 Temporal Memory Scoring and Synchronization........ 43
5. IMPLEMENTATION........ 44
5.1 System Initialization and Layered Configuration........ 45
5.2 The BYOM Provider Registry and State Management........ 46
5.2.1 Managing Provider-Specific Idiosyncrasies........ 46
5.2.2 Dynamic Credential Synchronization........ 46
5.3 The Composable Middleware Pipeline........ 47
5.3.1 The LIFO "Wrap" Execution Pattern........ 47
5.3.2 Systemic Throttling and Data Compaction........ 48
5.3.3 Framework vs. SandBot Middlewares........ 48
5.4 High-Frequency Frontend Streaming Architecture........ 49
5.4.1 Rendering Optimization via requestAnimationFrame (rAF)........ 49
5.4.2 IDE Deep-Linking and Generation Counters........ 49
5.5 Security Implementation and Infrastructure Lifecycle........ 50
5.5.1 The Three-Layer Provisioning........ 50
5.6 Environment, Observability, and Scaling Stack........ 51
5.6.1 Monitoring, Tracing, and Audit Logs........ 51
6. EVALUATION........ 52
6.1 Experimental Setup and Baselines........ 53
6.1.1 Environment and Telemetry Infrastructure........ 53
6.1.2 The Benchmark Task: Multi-Tier Deployment........ 54
6.1.3 Establishing the Monolithic Baseline........ 54
6.2 Efficiency, Token Optimization, and Cost (RQ2)........ 55
6.2.1 Impact of Tool Description Deduplication........ 55
6.2.2 The Compaction Middleware Pipeline........ 56
6.3 Evaluation of System Resilience and Scaling (RQ3, RQ4)........ 57
6.3.1 Simulated API Rate-Limit Recovery........ 57
6.3.2 Validating the Automated Provisioning........ 57
6.4 Effectiveness of Balanced Cooperation and Memory (RQ1)........ 59
6.4.1 Peer-to-Peer Task Decomposition........ 59
6.4.2 Long-Term Context Retention via Graphiti........ 59
6.5 Frontend Observability and Rendering Performance........ 60
6.5.1 Rendering Performance via rAF Batching........ 60
6.5.2 State Synchronization and Generation Counters........ 60
7. UI PRESENTATION........ 61
7.1 Projects page........ 62
7.2 Orchestrator chat page........ 63

7.3 Agents chat view page ... 66
7.4 Logs page ... 67
7.5 VMs page ... 68
7.6 Agents page ... 69
7.7 Workspace page ... 71
7.8 Tools page ... 72
7.9 Settings page ... 74
8. CONCLUSION AND FUTURE WORK ... 76
8.1 Thesis Summary and Synthesis of Findings ... 77
8.2 Limitations and Critical Analysis ... 78
8.3 Future Work and Research Directions ... 79
REFERENCES ... 81
APPENDICES ... 83
Appendix A: Atomic Agent Configuration ... 83
A.1 Role YAML (role.yaml) ... 83
A.2 Tool YAML (tools.yaml) ... 83
Appendix B: Tool Description Deduplication (MCP) ... 84
B.1 Raw JSON Schema (Abridged) ... 84
B.2 Deduplicated "Pointer" Format ... 84
Appendix C: Sample Execution Traces ... 85
C.1 Orchestrator State Transition (Arize Phoenix Extract) ... 85
C.2 Specialist Execution Log (Middleware Interception) ... 85
Appendix D: The Six-Section Prompt Template ... 86
D.1 Context Construction Template (Markdown) ... 86
Appendix E: Inter-Agent Communication Protocol ... 87
E.1 Task Delegation Envelope ... 87
Appendix F: OpenNebula Lifecycle State Mapping ... 88
Appendix G: Graphiti Memory Extraction Sample ... 89

List of Figures

- **Figure 1.1:** Transition from Monolithic LLM to SandBot Architecture.
- **Figure 3.1:** Sequence Diagram of the Orchestrator-Delegate Pattern.
- **Figure 3.2:** SandBot High-Level Macro-Architecture.
- **Figure 3.3:** VM Lifecycle State Machine.
- **Figure 6.1:** Token Comparison Chart (SandBot vs. Baseline).
- **Figure 6.2:** Reliability Heatmap: Success Rate vs. Latency.

List of Images

- **Image 7.1:** Projects page.
- **Image 7.2:** Orchestrator chat page. Completed tasks.
- **Image 7.3:** Orchestrator chat page. Delegation to other agents.
- **Image 7.4:** Orchestrator chat page. WAIT system interrupt.
- **Image 7.5:** Orchestrator chat page. Concurrent requests.
- **Image 7.6:** Agents chat view page. Communication between 2 agents.
- **Image 7.7:** Logs page.
- **Image 7.8:** VMs Dashboard page.
- **Image 7.9:** Agents page.
- **Image 7.10:** Edit Agent modal.
- **Image 7.11:** Workspace page.
- **Image 7.12:** Workspace page. Researcher chat.
- **Image 7.13:** Tools page.
- **Image 7.14:** Gateway VM provision modal.
- **Image 7.15:** Install Tool modal.
- **Image 7.16:** Project Settings page.
- **Image 7.17:** Project Settings page. LLM Profile and Model Visibility settings.
- **Image 7.18:** Project Settings page. Conversation Summarization and Context Budget settings.

List of Abbreviations

- **ACE:** Arbitrary Code Execution
- **BYOM:** Bring-Your-Own-Model
- **DCG:** Directed Cyclic Graph
- **HITL:** Human-in-the-Loop
- **KVM:** Kernel-based Virtual Machine
- **LIFO:** Last-In, First-Out
- **LLM:** Large Language Model
- **MAS:** Multi-Agent System
- **MCP:** Model Context Protocol
- **MicroVM:** Minimalist Virtual Machine
- **RAG:** Retrieval-Augmented Generation
- **ReAct:** Reasoning and Acting
- **TCO:** Total Cost of Ownership

1. INTRODUCTION

Recent advancements in Large Language Models (LLMs) have facilitated a paradigm shift from simple, single-turn conversational agents to highly capable, autonomous Multi-Agent Systems (MAS). As these foundational models improve in reasoning, zero-shot planning, and tool utilization, there is a growing imperative to apply them to complex, multi-step domains such as software engineering and infrastructure provisioning. However, moving from theoretical benchmarks to real-world, stateful execution exposes severe practical limitations in current orchestration frameworks. To address these architectural bottlenecks, this thesis introduces **SandBot** (an Agent Sandboxing & Orchestration Framework), an advanced system designed to coordinate specialized AI agents through balanced peer-to-peer cooperation, dynamic virtualized sandboxing, and highly extensible tooling.

1.1 Motivation

The standard approach to AI-driven automation currently relies on a single, monolithic LLM to handle all aspects of a task - from high-level planning to granular code execution. While these unified agents perform exceptionally well on isolated, short-horizon problems, applying them to intricate, long-running projects introduces significant structural challenges [8].

The primary barrier to multi-step autonomous execution is the mathematical and physical limit known as the "memory wall." Because the self-attention mechanisms powering modern LLM architectures scale quadratically ($O(n^2)$), simply expanding the context window to ingest entire project histories results in untenable computational overhead and latency [16]. As a monolithic agent iteratively works through a complex project, it accumulates a massive, noisy history of API tool outputs, stack traces, error logs, and intermediate reasoning steps.

This continuous accumulation leads directly to severe context degradation. Empirical research proves that as context windows fill, models experience a U-shaped "lost-in-the-middle" effect: they retain a strong primacy bias (remembering the initial prompt) and a recency bias (remembering the last error), but effectively forget constraints, instructions, and architectural decisions buried in the center of their context [13], [14]. This cognitive degradation predictably leads to hallucinated outputs and a gradual drift from the user's original objective.

Compounding this cognitive issue is the economic and operational tendency of the current AI ecosystem to tightly couple orchestration logic with the proprietary APIs of specific model providers (e.g., OpenAI, Anthropic). This vendor lock-in restricts architectural flexibility and generates severe long-term liabilities. Organizations relying solely on proprietary models face "institutional lock-in" - where the workforce loses the foundational expertise required to verify automated outputs [19]. Furthermore, agentic workflows intrinsically require repeated reasoning loops and error-correction cycles. When executed entirely through commercial APIs, the high volume of necessary retries acts as an "unreliability tax" that severely inflates the Total Cost of Ownership (TCO) for autonomous systems [18].

Finally, as these autonomous agents are granted the agency to execute code, modify file systems, or provision cloud services, they introduce substantial security vulnerabilities. Allowing an AI to run generated scripts without strict, hardware-level isolation poses a direct threat to host systems, frequently exposing them to Arbitrary Code Execution (ACE) via prompt injection or unexpected dependency chains [1]. Research in Automatic Exploit Generation (AEG) has demonstrated that vulnerability identification and control-flow hijacking can be formalized and executed at machine speed [7]. This technical reality necessitates that **SandBot** moves beyond superficial software firewalls, instead utilizing **OpenNebula** to provision ephemeral, network-isolated MicroVMs as a verifiable lifecycle guarantee. Therefore, there is a critical need to transition away from isolated, context-heavy monolithic agents toward secure, specialized, cooperative, and model-agnostic multi-agent networks.

1.2 Problem Statement

Despite the theoretical benefits of Multi-Agent Systems, existing orchestration frameworks (e.g., AutoGen, CrewAI) often lack the production-ready features, memory management, and security boundaries necessary for complex technical execution. Specifically, current systems struggle with four key architectural gaps:

1. **Rigid Topologies and Hub Competence Bottlenecks:** Current frameworks frequently impose rigid, hierarchical topologies. Quantitative studies into the "Science of Scaling" reveal that centralized hub-and-spoke models suffer from critical "hub competence" bottlenecks: if the central orchestrator hallucinates or fails, the error is amplified downstream [24], [25]. Consequently, there is a lack of frameworks facilitating balanced, peer-to-peer cooperation capable of managing the "tool-coordination trade-off" [24].

2. **Absence of Ephemeral, Hardware-Level Sandboxing:** There is a distinct lack of secure execution environments built natively into the orchestration loop. Most frameworks execute agent-generated code locally or in Docker containers, which share the host kernel and are acutely vulnerable to breakout attacks [4]. They fail to utilize dynamically provisioned, network-isolated sandboxes mapped specifically to the agent's lifecycle [5].

3. **Context Exhaustion via Extensible Tooling:** Managing the context window remains difficult as toolsets expand. Loading multiple, complex API schemas simultaneously accelerates context exhaustion. While protocols like the Model Context Protocol (MCP) [9] standardize tool access, injecting massive JSON schemas for infrastructure management (e.g., OpenNebula) quickly overwhelms the model, highlighting the need for dynamic, intent-based tool filtering and algorithmic deduplication [10].

4. **Fragility in Distributed Orchestration:** Distributed AI orchestration is inherently vulnerable to network timeouts, rate limits, and sub-service failures. Many current frameworks lack resilient, "fail-soft" architectures that can gracefully degrade (e.g., falling back to process-local memory if a Redis message broker fails) without crashing the entire agentic loop.

1.3 Research Questions

To address the problems outlined above, this thesis formulates and investigates the following primary research questions (RQs):

- **RQ1:** How can a peer-to-peer, Orchestrator-Delegate MAS topology, combined with distinct memory namespaces, mitigate context degradation and "lost-in-the-middle" hallucination in multi-step software deployments compared to current baselines?

- **RQ2:** To what extent does standardizing tool access via the Model Context Protocol (MCP) - augmented by algorithmic Tool Description Deduplication - reduce token overhead and API latency during complex infrastructure orchestration?

- **RQ3:** How can ephemeral, virtualized sandboxing (via OpenNebula MicroVMs) be programmatically integrated into the stateful agent lifecycle to prevent Arbitrary Code Execution (ACE) without introducing prohibitive execution latency?

- **RQ4:** How does decoupling the orchestration framework from the cognitive engine via a Bring-Your-Own-Model (BYOM) architecture impact system resilience and operational Total Cost of Ownership (TCO)?

1.4 Objectives and Scope

The primary objective of this thesis is to design, implement, and rigorously evaluate the **SandBot** framework to answer the established research questions. A core architectural goal is the **explicit decoupling of the underlying Framework (the cognitive engine)** from the **SandBot orchestration logic (the project / task decomposition strategy)**. By structurally separating execution primitives from domain-specific agent roles, **SandBot** aims to provide a stable, economically viable, and highly secure environment for executing complex projects.

The scope of this research focuses on developing a cooperative network of specialized agents. To support this, the system integrates the following core components:

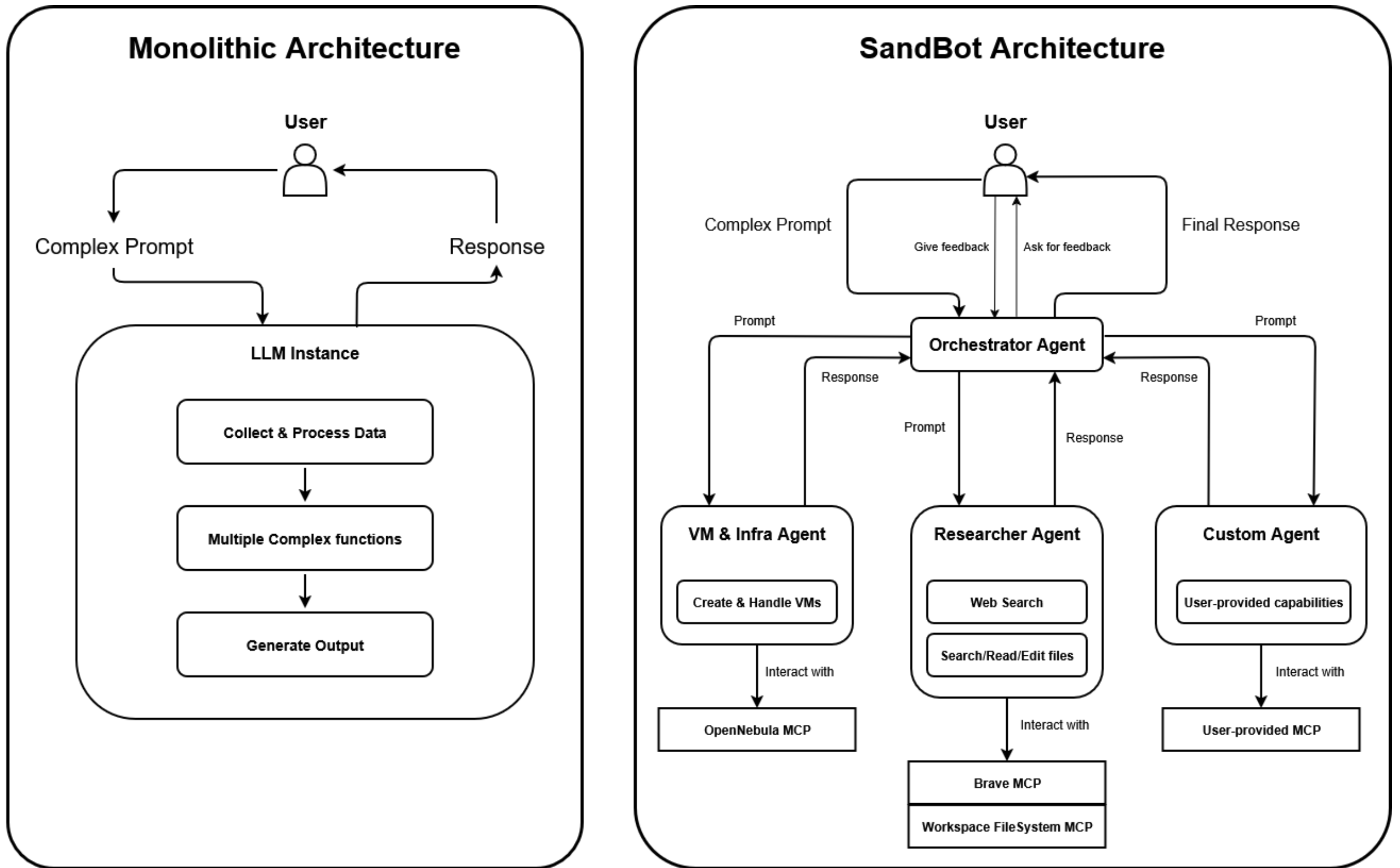
- **Framework Layer (The Cognitive Engine):** A specialized library built on LangGraph that provides the low-level primitives for state management, memory namespaces, LIFO middleware execution, and hardware-isolated tool-calling endpoints.

- **SandBot Layer (The Orchestration Architecture):** The specific implementation of the Orchestrator-Delegate pattern, defining how specialized "Atomic Agents" coordinate to solve infrastructure and software engineering tasks using the Framework's primitives.

- **Model Context Protocol (MCP):** Serves as a universally standardized mechanism to dynamically extend the agents' toolsets.

- **OpenNebula Lifecycle Integration:** Automatically provisions virtualized execution environments [2] using a "Three-Layer Provisioning" approach.

- **Bring-Your-Own-Model (BYOM) Architecture:** Ensures the system remains model and provider agnostic, seamlessly routing tasks between multiple proprietary cloud APIs and locally hosted open-sourced models.



[FIGURE 1.1: High-level conceptual diagram showing the transition from a monolithic LLM script to the SandBot Multi-Agent Orchestration architecture.]

1.5 Summary of Contributions

This thesis provides theoretical, architectural, and practical engineering contributions to the field of autonomous AI orchestration. Specifically, it introduces:

1. **Structural Decoupling of Agent Frameworks:** Anarchitectural split between a generic "Agent Framework" (handling execution mechanics) and the "SandBot" layer (handling task decomposition, enabling modularity and model-agnosticism).

2. **The Four-Component Atomic Agent:** A standardized agent anatomy featuring immutable core logic combined with runtime-mutable user rules, balancing flexibility and safety.

3. **Context & Token Optimization Algorithms:** The implementation of "Tool Description Deduplication" alongside proprietary mathematical memory scoring formulas.

4. **High-Frequency Frontend Observability:** A novel useAgentStream React hook utilizing requestAnimationFrame (rAF) batching to render massive parallel token streams in realtime.

5. **The Fail-Soft MAS Architecture:** An infrastructure-resilient design capable of graceful degradation, ensuring multi-agent state machines survive peripheral sub-service outages (for example mcp-tools, VMs, LLM provider endpoints, etc.).

1.6 Thesis Structure

The remainder of this document is structured as follows:

- **Chapter 1 (Introduction):** Presents the scope of this thesis, from the original motivation to the problems it tries to address.
- **Chapter 2 (Background and Related Work):** Reviews the foundational literature on MAS orchestration topologies, the evolution of tool standardization (MCP), the necessity of secure virtualized sandboxing, and the economic imperatives & future-proofing of model-agnostic (BYOM) architectures. It includes a comparative analysis against state-of-the-art frameworks.
- **Chapter 3 (Overview of SandBot):** Presents the high-level macro-architecture, detailing the modular three-layer design (Frontend, Backend, Framework), the Orchestrator-Delegate pattern, and the fail-soft infrastructure resilience methodologies.
- **Chapter 4 (Methodology):** Explores the theoretical algorithms driving the system, the Six-Section Prompt architecture, and active versus passive RAG memory strategies.
- **Chapter 5 (Implementation):** Details the concrete software engineering decisions, including the composable middleware pipelines, API provider intricacies management, and the programmatic integration of OpenNebula using automatic provisioning.
- **Chapter 6 (Evaluation):** Provides an empirical analysis establishing quantitative baselines for token efficiency, system resilience, and task completion success rates against monolithic baselines during a simulated complex application deployment.
- **Chapter 7 (UI Presentation):** Showcase of the implementation’s frontend, as a guide to the expected user flows, describing the end-user interaction with the entire system.
- **Chapter 8 (Conclusion and Future Work):** Synthesizes the core findings, outlines current system limitations, and proposes actionable directions for future autonomous AI coordination research.

2. BACKGROUND AND RELATED WORK

To contextualize the architectural design and theoretical-practical contributions of the SandBot framework, it is essential to examine the foundational technologies and existing literature that informed its development. This section reviews the rapid evolution of Multi-Agent Systems (MAS), the standardization of tool access via the Model Context Protocol (MCP), the necessity of secure hardware-level execution environments, advanced memory management strategies, and the growing importance of model-agnostic architectures. Finally, it provides a comparative analysis of state-of-the-art orchestration frameworks to explicitly identify the research gaps that SandBot addresses.

2.1 Multi-Agent Systems (MAS) and Stateful Orchestration

The architectural deployment of Large Language Models (LLMs) has aggressively evolved from isolated, zero-shot conversational interfaces toward sophisticated, autonomous Multi-Agent Systems (MAS). This evolution is driven by the realization that monolithic models - regardless of their parameter count - frequently encounter cognitive bottlenecks when processing long-spanning, multi-domain tasks or maintaining strict role consistency across diverse sub-disciplines [8].

2.1.1 The Convergence of Reasoning and Action (ReAct)

The foundational logic underpinning early agentic workflows is the ReAct (Reasoning and Acting) paradigm. ReAct resolves the historical decoupling of internal model deliberation from external environmental interaction. It enables LLMs to generate reasoning traces (internal thoughts) and task-specific actions (tool calls) in an interleaved / iterative loop [22]. In contrast to standard Chain-of-Thought (CoT) prompting, which often propagates associative errors due to its reliance purely on static parametric knowledge, ReAct grounds the agent's reasoning in real-time observations retrieved from external APIs [22]. However, while ReAct successfully manages discrete, single-agent loops, it lacks the necessary topological scaffolding for complex, multi-agent project management.

2.1.2 From Rigid Hierarchies to Graph-Centric State Machines

To scale agentic capabilities beyond a single loop, first-generation MAS frameworks introduced conversational and organizational topologies. Frameworks such as MetaGPT enforce rigid, hierarchical structures inspired by human corporate workflows (e.g., assigning rigid roles like "Product Manager" and "Software Architect"). These systems rely heavily on Standardized Operating Procedures (SOPs) and "Publish-Subscribe" communication protocols to coordinate tasks [23].

While highly structured, hierarchical systems suffer from systemic "hub competence" vulnerabilities: if a central orchestrator model fails, hallucinates, or misinterprets an SOP, the error is amplified disproportionately downstream [24], [25]. Historically, pre-AI autonomous cybersecurity bots, like Mayhem, demonstrated the efficacy of hybrid cyber reasoning and Bayesian-prioritized decision making [11]. Mayhem prioritized remediation efforts based on calculated exploitability, necessitating a stateful control flow seen in modern directed cyclic graphs. This concept establishes a baseline precedent for autonomous MAS, directing attention to the stateful aspect of agentic engines.

Due to the expected benefits, the field has shifted toward stateful, graph-centric orchestration, exemplified by foundational libraries like LangGraph. LangGraph transitions away from rigid, linear chains toward Directed Cyclic Graphs (DCGs), where nodes represent specialized agents and edges define stateful control flow. This architecture allows for persistent memory across loops, complex conditional branching, and robust error recovery through built-in state checkpointing [28].

2.1.3 Topologies and the Science of Scaling

The structural topology of a multi-agent system fundamentally dictates some core attributes, such as its resilience, token efficiency, and cognitive accuracy. Quantitative studies into the "Science of Scaling" for agentic systems reveal a crucial "Tool-Coordination Trade-off" [24]. Centralized (hub-and-spoke) topologies excel at highly parallelizable, simple tasks but introduce severe cognitive bottlenecks at the leader node. Conversely, decentralized (peer-to-peer) mesh networks demonstrate superior flexibility and consensus resilience for dynamic, open-ended tasks [26].

Furthermore, biologically inspired AI coordination - exemplified by the **AIRS-x** variant - introduces the concept of **clonal selection and affinity maturation** to maintain model diversity [6]. By utilizing stimulation thresholds in a competitive agent environment, AIRS-x ensures that only the most high-performing specialists survive within resource-constrained systems. This competitive mechanism provides a theoretical basis for SandBot's resource allocation logic in virtualized playgrounds.

However, scaling principles reveal a "Capability Saturation" effect: blindly injecting more agents into a sequential workflow often degrades overall performance due to inter-agent communication overhead and severe information fragmentation [24]. Therefore, modern frameworks must focus on strict separation of concerns and session isolation mechanisms - where inter-agent communication is partitioned - to minimize the coordination tax and prevent context contamination.

2.2 Extensible Tooling and the Model Context Protocol (MCP)

For an autonomous AI agent to affect change in the real world, it requires programmatic access to external tools, APIs, and file systems. Historically, integrating these capabilities required bespoke, tightly coupled middleware tailored to the specific function-calling formats (e.g., JSON schemas) of individual LLM providers. This market fragmentation made scaling multi-agent systems exceptionally difficult, recreating the classic $M \times N$ integration problem of software interoperability: every new AI host requires a custom connector for every existing tool [12].

The recent introduction of the Model Context Protocol (MCP) [9] represents a pivotal standardization effort in resolving this integration bottleneck between different ecosystems. Building upon the interoperability paradigms established by the Language Server Protocol (LSP) for human developers, MCP provides a universal, open standard for connecting AI models to external data sources and infrastructure [10]. By adopting a strict Client-Host-Server architecture, with common communication rules, MCP allows developers to build isolated, reusable tool servers that expose specific capabilities over standard communication channels (e.g., stdio, SSE).

A crucial innovation of the MCP is its definition of core communication primitives: Resources (for static data retrieval), Tools (for model-initiated execution), and Prompts (for predefined workflow templating) [9]. An agent acting as an MCP client can dynamically connect to these servers at runtime, seamlessly discovering available tools and their required schemas / rules / data expectations, without prior hardcoding.

However, while this standardization solves the interoperability bottleneck, it introduces novel supply-chain security risks. As agents implicitly trust connected MCP servers, malicious actors can exploit the protocol via "tool poisoning" or silent redefinition attacks to inject harmful instructions directly into the agent's context window [29]. Consequently, robust architectures must treat MCP extensions not only as functional utilities, but as active security boundaries [1].

2.3 Dynamic Virtualized Playgrounds and Security

As autonomous agents are granted extensible tooling - particularly for software development, script execution, and infrastructure management - system security becomes a paramount concern. When an AI agent is tasked with writing and executing code iteratively, it introduces the direct risk of Arbitrary Code Execution (ACE). Whether through a model hallucination, an unexpected dependency chain, user error, or a malicious prompt injection embedded in RAG-retrieved data, an agent can effortlessly generate destructive shell commands [1].

Historically, standard containerization technologies, such as Docker, have been used to isolate agent execution. However, Docker containers fundamentally share the host operating system's kernel, which is a valid point of concern. While they provide basic namespace isolation, they remain critically vulnerable to privilege escalation and container breakout attacks, such as the widely documented "Leaky Vessels" runc vulnerabilities [4]. This shared-kernel architecture poses an unacceptable risk when deploying highly autonomous agents in enterprise production environments.

To achieve true, hardware-level isolation, there is a growing necessity for dynamic virtualized playgrounds: ephemeral Virtual Machines (MicroVMs) provisioned specifically for the duration of a single task or session [5]. While traditional VMs have historically suffered from massive provisioning latency that made them unsuitable for rapid API responses, recent programmatic infrastructure management platforms, such as OpenNebula, enable the automated, near-instantaneous deployment of these network-isolated environments utilizing hypervisor technology like KVM [2], [3]. By mapping the lifecycle of a secure MicroVM directly to the lifecycle of an agentic workflow, systems can enforce a strict "fail-closed" security posture. Even if an agent generates highly destructive commands or attempts lateral network movement, the blast radius is strictly contained within the ephemeral VM [3], [5].

2.4 Context Optimization and Stateful Memory Architectures

As the autonomy of AI agents expands, the primary constraints on intelligence are no longer merely parameter counts, but the fundamental mechanics of memory persistence and context management. Expanding the "effective" context window of these systems has necessitated a radical departure from simple memory ingestion toward modular, hierarchical context structures [30].

2.4.1 The Cognitive Taxonomy of LLM Memory

To architect more effective memory systems, researchers have adapted cognitive psychology taxonomies - specifically Endel Tulving's biological memory systems - to LLM architecture [30]. In this mapping:

- **Semantic Memory** represents the general world knowledge implicitly stored within the model's parametric weights post-training.
- **Procedural Memory** encompasses the system's instructions, learned behaviors, and system prompts that dictate persona and formatting rules.
- **Episodic Memory** acts as the working memory, usually handled by the active context window or Key-Value (KV) cache, tracking the specific events of a conversation or task execution [30], [16].

Because episodic memory is momentary and resource constrained, a major focus of multi-agent orchestration is externalizing this working state into persistent, query-able memory layers.

2.4.2 The Mathematical and Empirical "Memory Wall"

The quintessential challenge in managing episodic memory lies in the mathematical complexity of the self-attention mechanism. Because attention scores scale quadratically ($O(n^2)$) relative to sequence length, the memory footprint required for KV cache activations during long-context inference will frequently exceed the available hardware bandwidth [16].

However, even when hardware permits massive context windows (e.g., 1M+ tokens), empirical research reveals severe cognitive vulnerabilities. In an extensive study, Liu et al. identified the "lost-in-the-middle" phenomenon: a persistent degradation in LLM reasoning when critical information is buried near the center of a long prompt [13].

Modern solutions to these limitations involve structured grounding. The **LYRICEL** framework established a conceptual ancestor to SandBot by integrating Knowledge Graphs (KG) with LLMs to ensure explainable results [15]. By grounding generative models in factual foundations, LYRICEL mitigates context degradation in high-dimensional tasks. This approach provides the theoretical basis for SandBot's **Six-Section Prompt Architecture**, which operationalizes structured grounding to overcome the "memory wall" [15].

2.4.3 External Memory: Vector RAG versus GraphRAG

To combat context exhaustion and positional bias, advanced frameworks implement Retrieval-Augmented Generation (RAG) to inject relevant knowledge strictly on an as-needed basis. Traditional Vector RAG converts unstructured text into dense embeddings and retrieves chunks based on semantic similarity. While highly scalable, vector retrieval flattens structural context and struggles profoundly with "multi-hop" reasoning [17].

Consequently, the field is rapidly shifting toward GraphRAG, which models knowledge as a structured network of entities (nodes) and explicit relationships (edges) [17]. Instead of merely measuring cosine distance, GraphRAG traverses semantic edges to assemble a relationship-aware context.

2.5 The Sovereign AI Imperative: BYOM and Open-Weight Architectures

The global transition from generative AI experimentation toward scaled enterprise production has exposed a structural vulnerability in early framework designs: the reliance on single-provider, closed-source models. Building an orchestration framework tightly coupled to a proprietary API introduces multi-layered vendor lock-in [21], transforming technical inconvenience into a hidden liability [20].

2.5.1 The Dimensions of Vendor Lock-in and the "Unreliability Tax"

Proprietary APIs strictly dictate data formatting, personas, function-calling schemas, and context limits. Furthermore, organizations blindly automating logic through "black-box" proprietary models face a severe erosion of human capital - "institutional lock-in" - where the workforce loses the foundational expertise required to verify automated outputs [19].

Most critically, agentic workflows intrinsically require repeated reasoning loops, often tool-calling retries, and reflections. When executing these multi-step loops entirely through commercial APIs, the cost per solution will scale exponentially. High error rates and necessary iterations act as an "unreliability tax" that severely inflates Total Cost of Ownership (TCO) [18].

2.5.2 Economic Routing and Hybrid Execution (BYOM)

To mitigate these risks and support "Sovereign AI", modern orchestration architectures increasingly mandate the Bring-Your-Own-Model (BYOM) paradigm. This approach structurally decouples the core orchestration logic from the underlying cognitive engines.

TCO mathematical models demonstrate that self-hosting large models becomes economically superior to managed APIs primarily when daily token throughput crosses massive thresholds [18]. Therefore, the optimal operational strategy is hybrid routing. A model-agnostic framework can dynamically route tasks based on cognitive complexity: assigning complex architectural planning to a large-parameter proprietary cloud model, while delegating simpler, repetitive formatting tasks and intermediate reflection loops to smaller, highly quantized open-weight models running locally (e.g., via Ollama). The user can define themselves different provider profiles, including configuration and options such as cooldown between requests, as well as LLM profiles, which combine providers, models and default options for each model (such as temperature or helpful default prompts).

2.6 Comparative Analysis of Current Orchestration Frameworks

To clearly delineate the research gaps addressed by SandBot, it is necessary to compare it against the current State-of-the-Art (SOTA) in Multi-Agent System orchestration. While numerous open-source libraries exist, the landscape is dominated by a few distinct architectural approaches.

1. **CrewAI:** Built on top of LangChain, CrewAI utilizes a rigid, role-playing sequential/hierarchical topology. It operates on "processes" (Sequential or Hierarchical) where tasks are passed linearly. While it provides excellent developer experience (DX) for predictable, static workflows, its rigid structure struggles with dynamic, fail-soft resilience and complex asynchronous tool orchestration. Like AutoGen, it lacks dynamic, hypervisor-level sandboxing integration.

2. **Microsoft Agent Framework:** Based on Semantic Kernel & AutoGen, it is a highly popular framework built primarily around conversational multi-agent patterns. Agents interact via simulating human group chats. While excellent for brainstorming and zero-shot code generation, its unstructured conversational topology frequently leads to infinite conversational loops, massive prompt contamination, and rapid susceptibility to the "lost-in-the-middle" phenomenon. It lacks native hardware-level sandboxing, relying predominantly on local Docker execution [8].

3. **SWE-agent:** A domain-specific framework designed explicitly for software engineering (resolving GitHub issues). SWE-agent pioneered the concept of an Agent-Computer Interface (ACI), customizing bash environments to be more LLM-friendly (e.g., formatting tool outputs) [1]. While highly effective at its specific niche, it is not a generalized framework and does not natively support extensible interoperability protocols like MCP for broad infrastructure orchestration.

4. **LangGraph:** Developed by LangChain, LangGraph is a low-level library for building stateful, multi-actor applications with LLMs, modeling workflows as cyclical graphs [28]. It provides the mathematical and state-management primitives required for advanced orchestration.

The SandBot Differentiation: SandBot does not replace low-level graph management; rather, it utilizes LangGraph as its foundational state-machine engine (Chapter 3). The core gap in current research and tooling is the **Application Layer for Secure Orchestration**. Frameworks like AutoGen and CrewAI focus heavily on prompting techniques, while displacing the need for infrastructure safety and context overhead. SandBot fills this gap by introducing:

1. Native integration with programmatic hypervisors (OpenNebula) for true hardware-level sandboxing.

2. Advanced algorithmic context compaction (Tool Description Deduplication) to specifically manage the token overhead introduced by standardized MCP integrations.

3. A strict "Four-Component Atomic Agent" model governed by peer-to-peer task delegation, directly solving the unconstrained context exhaustion seen in traditional MAS frameworks.

3. OVERVIEW OF SANDBOT (MACRO-ARCHITECTURE)

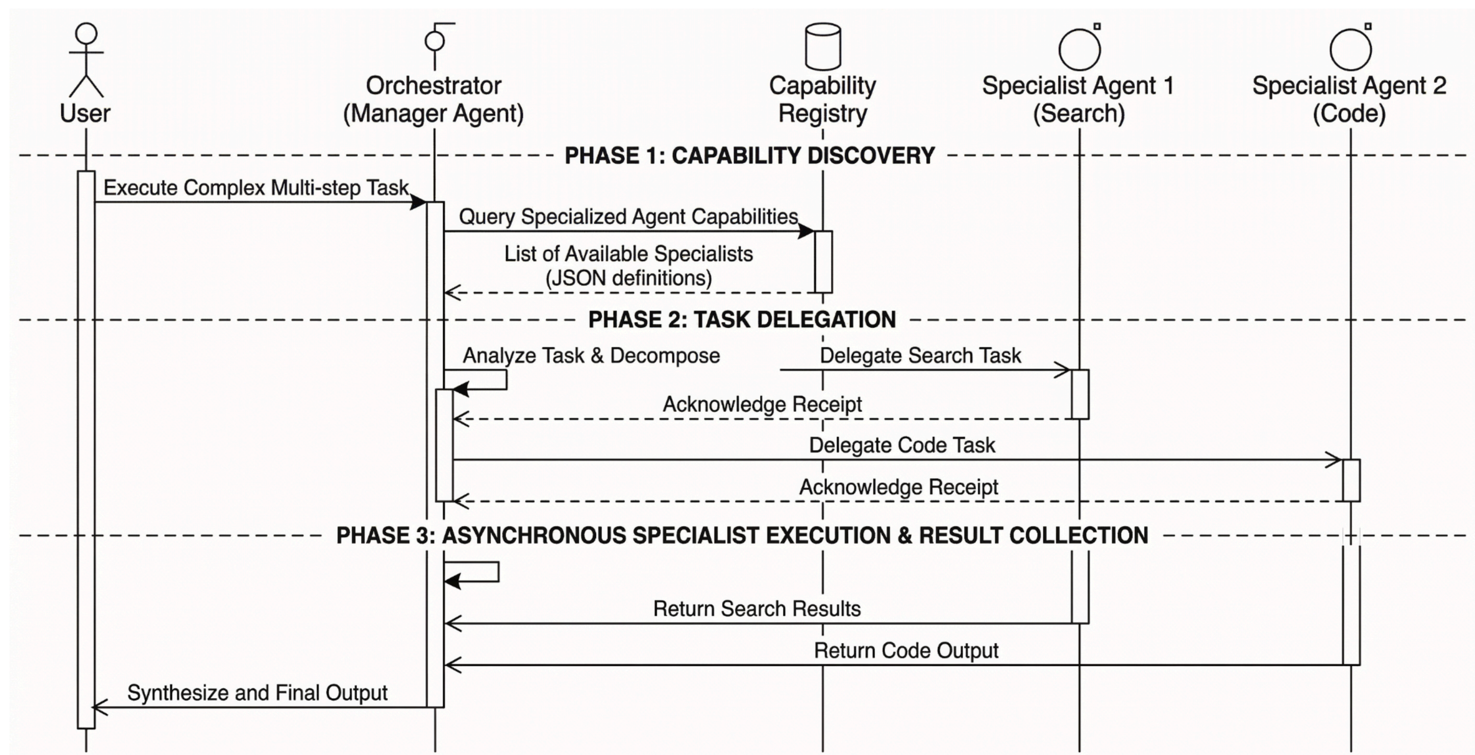
Drawing upon the theoretical foundations of stateful orchestration, secure sandboxing, and the Bring-Your-Own-Model (BYOM) paradigm established previously, this section introduces SandBot. SandBot is engineered as a robust, three-layer distributed system designed to bridge the gap between experimental multi-agent research and high-performance, enterprise-grade deployment. This chapter defines the system's conceptual model and outlines its macro-architecture. Details will be presented regarding the infrastructure resilience, middleware pipelines, and security-first principles that attempt highly deterministic execution in unpredictable environments.

3.1 Conceptual Model: The Orchestrator-Delegate Pattern

At its core, SandBot defines "successful" complex project execution not merely as the generation of accurate code or text, but as the deterministic, verifiable, transparent, and secure application of an autonomous workflow to an external environment. In this architecture, we distinguish between the **Framework** - which provides the "Cognitive Engine" - and **SandBot** - which represents the specific orchestration strategy and agent network built upon it.

SandBot rejects traditional hub-and-spoke topologies that inherently create cognitive bottlenecks [24]. Instead, it relies on a sophisticated **Orchestrator-Delegate Pattern** governed by a strict separation of concerns. When an unstructured natural language query enters the system, it is intercepted by a primary Orchestrator. However, the Orchestrator is explicitly forbidden from directly accessing infrastructure tools or executing code. Instead, it functions purely as a user-facing semantic router and high-level strategist, while also maintaining the human-in-the-loop model.

Orchestrator-Delegate Pattern: Asynchronous Parallel Execution



[FIGURE 3.1: Sequence Diagram of the Orchestrator-Delegate Pattern, showing the capability discovery phase, task delegation, and asynchronous specialist execution.]

Utilizing a dynamic "Capability Discovery" methodology, the Orchestrator assesses available specialist agents at the beginning of each request. For example, it might identify the vm_ops_specialist for interacting with the OpenNebula MCP server, or the research_specialist for web-scale information retrieval. It communicates its strategic intent using a mandatory structured output format encompassing three distinct streams:

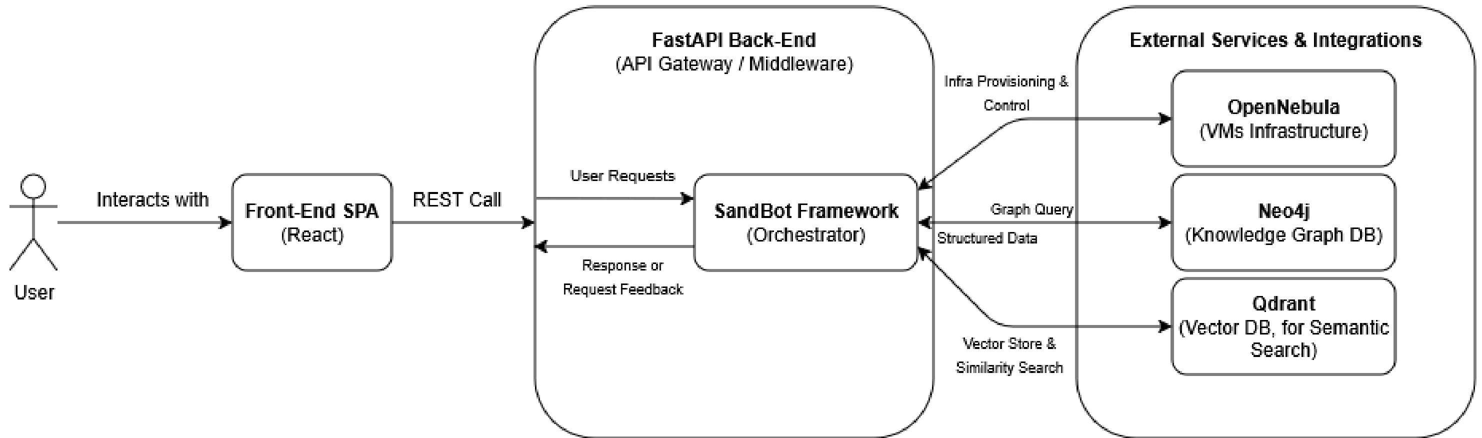
- **$THOUGHTS:** The agent's internal reasoning, hypothesis generation, and strategic planning. This stream is hidden from the final user output but is captured for system tracing and debugging.
- **$USER:** The final, polished message delivered to the human user.
- **$ACTION:** Deterministic system directives (e.g., WAIT, ASK, DONE) commanding the underlying framework to transition states.

The order of the 3 markers is quite important as it indirectly “forces” the agent to reason before acting, and to hold accountability (towards the user / caller) for its actions before performing them. This was experimentally proved to provide better guidance during task completion and removed many “impulsive” actions that would be taken otherwise. To avoid deviating from its current objectives, every agent is encouraged to create and use a to-do list structure, which can also serve as ephemeral note pad. These to-do lists can be viewed or even edited by the user, in favor of transparency and re-routing the current path taken by the agent. The Orchestrator delegates granular objectives to other specialists via a standardized task tool, engaging in a peer-to-peer network where sub-agents execute operations autonomously.

To improve reliability, the Orchestrator enforces a "Pre-completion check." Before calling its final terminal function, the Orchestrator systematically reviews the active tasks and to-do lists to ensure that all dispatched tasks within the active registry have reached a terminal state (DONE or BLOCKED). This strictly enforced protocol eliminates the "silent-drop" failures commonly observed in unconstrained multi-agent loops.

3.2 The Three-Layer System Architecture

To support distributed, stateful intelligence without sacrificing maintainability or security, SandBot is constructed upon three distinct, decoupled operational layers.



[FIGURE 3.2: High-Level Macro-Architecture Diagram illustrating the Frontend SPA, the FastAPI Backend, the Framework cognitive engine, and the external integrations (OpenNebula, Neo4j, Qdrant).]

3.2.1 Frontend Layer (Observability and Presentation)

The user-facing presentation layer is implemented as a React Single Page Application (SPA). Beyond basic project lifecycle management, its primary architectural function is observability into the ReAct model (Reasoning and Acting) inner monologue [22]. The Frontend utilizes a highly optimized useAgentStream hook to maintain persistent, bidirectional WebSocket connections with the backend.

To handle the immense data volume generated by parallel LLM streams (frequently exceeding hundreds of tokens per second), the presentation layer implements requestAnimationFrame (rAF) batching. This ensures consistent UI rendering without blocking the browser's main thread. Furthermore, it incorporates strict generation (gen) counters to precisely synchronize incoming streams, preventing UI flicker and guaranteeing that stale data is never rendered following an asynchronous state restart.

3.2.2 Backend Layer (API Gateway and Session Management)

Serving as the intermediary service layer, the Backend is constructed using FastAPI. It acts as a traffic controller for the entire architecture, managing stateless REST API operations for metadata updates alongside bidirectional WebSockets for low-latency token streaming, interactive console access (SSH/VNC into provisioned MicroVMs) and realtime logging, providing transparency to the user.

The Backend is subdivided into specialized management services:

- **AgentService:** Manages agent instances, enforces idle timeouts to aggressively conserve memory, and handles hot-swapping logic for rate-limit event callbacks.
- **ProjectManager:** Handles secure, filesystem-backed CRUD operations for persistent projects.
- **OneService:** Specifically integrates with the OpenNebula CLI, parsing complex XML telemetry outputs into structured JSON to facilitate real-time infrastructure state tracking for the cognitive layer.

Crucially, the Backend integrates the orchestration logic via a dependency injection pattern (get_framework), ensuring centralized resource management. The framework is initialized explicitly during the FastAPI lifespan handler, guaranteeing that all agentic services, database connections, and background reconciliation tasks are operational before accepting external traffic.

3.2.3 Framework Layer (Stateful Cognitive Engine)

The foundational cognitive engine is a specialized internal library powered by LangGraph [28], modeling the multi-agent workflow as a Directed Cyclic Graph (DCG). It is critical to understand that this layer provides the "low-level" execution primitives, while **SandBot** provides the "high-level" specialized implementation.

The Framework Layer handles:

- **State Graph Management:** Modeling workflows as Directed Cyclic Graphs (DCGs).
- **Memory Primitives:** Managing the isolated namespaces (Private, Shared, Cross-Agent).
- **Tool Execution Hooks:** The pre_tool and post_tool interception points for security and compaction.
- **Context Assembly:** The mathematical logic of the Six-Section Prompt Architecture.

By isolating these primitives into a generic Framework, **SandBot** achieves its goal of **sovereign, model-agnostic execution**. The Framework doesn't "know" how to deploy a database; only how to manage an agent that does. This separation allows the same Framework to be reused for entirely different domains (e.g., medical research or legal analysis) simply by defining a new set of **SandBot** specialist agents.

It defines agents through a strict, four-part configuration schema designed around operational safety:

1. **Role YAML:** Defines the agent's core identity, associated LLM profiles, and memory namespace configurations. Identity-critical agents (e.g., infrastructure managers) are flagged as immutable: true to prevent unauthorized modification.

2. **Tool YAML:** Configures MCP server connections, transport protocols, and dynamic tool filtering rules.

3. **System Prompt:** A heavily parameterized Markdown template managed via the framework's structured prompt architecture.

4. **User Rules:** Runtime-mutable instructions that provide a customization layer without compromising architectural safety guarantees.

To definitively prevent context exhaustion, the Framework Layer utilizes a dynamic **Six-Section Prompt Architecture**. The ContextAssembler dynamically constructs the LLM context window by pulling data into six strictly governed partitions: System Prompt, Memory, RAG, Conversation, Tools, and Scratchpad. Each section is rigidly constrained by a specific "Budget Key" defined in the Role YAML, mathematically guaranteeing that prompt rehydration never exceeds the specific provider's token limits.

3.3 Infrastructure Resilience and Graceful Degradation

A fundamental reality of distributed AI orchestration is the inevitability of downstream infrastructure failures. SandBot relies on a comprehensive containerized stack: PostgreSQL (metadata and state checkpoints), Neo4j/Graphiti (graph-based memory), Redis (scaling/pub-sub message broker), Qdrant (vector retrieval), and Ollama (local open-weight inference).

To counteract infrastructure volatility, SandBot employs a "Fail-Soft" philosophy designed for graceful degradation. The framework executes continuous health checks, surfacing status indicators to the frontend that distinguish between core requirements and optional observability features.

If the system detects a failure in an optional component - for example, if the Redis pub/sub broker goes offline - the middleware pipeline dynamically bypasses the unresponsive service. The internal CommunicationMode seamlessly toggles from REDIS_PUBSUB (used for multi-pod Kubernetes horizontal scaling) to a local IN_PROCESS mode. This restricts inter-agent messaging to a single Python process via native asynchronous queues. While distributed horizontal scaling is temporarily degraded, this dynamic toggle guarantees that the active agentic loop remains uninterrupted despite peripheral system outages.

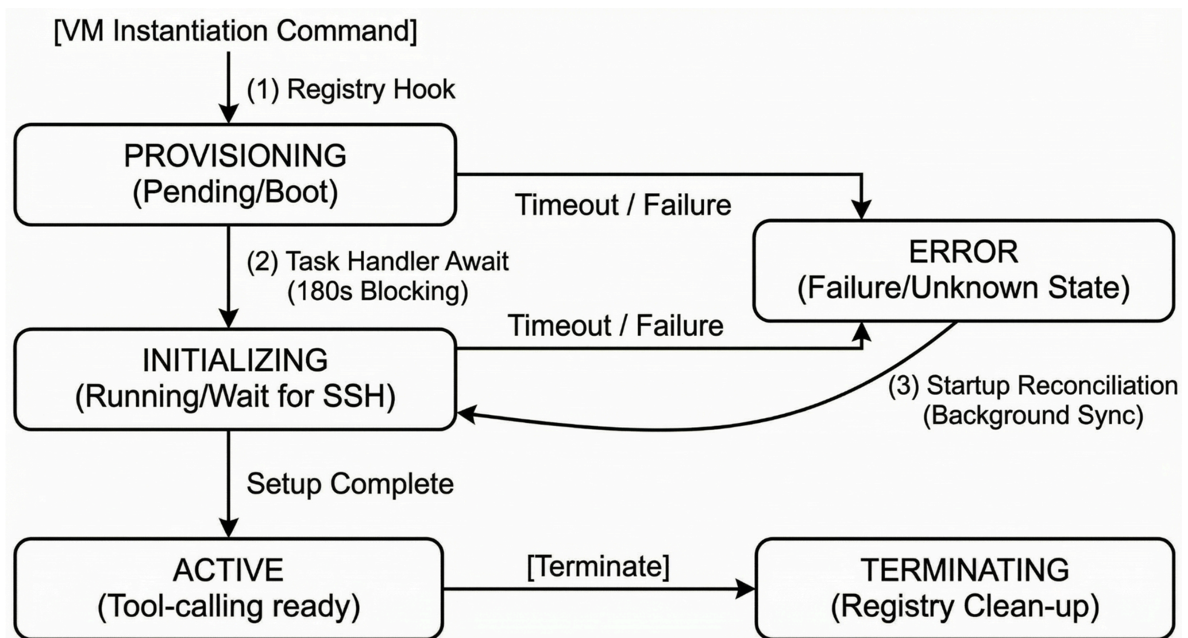
3.3.1 The Three-Layer Provisioning

When interfacing with highly sensitive cloud infrastructure via the OpenNebula MCP server (one-mcp), the framework enforces a "Three-Layer Provisioning" strategy to ensure virtual machines are registered and configured under a "fail-closed" policy. This was deemed necessary as arbitrary user or agent actions in the Virtual Machines can cause instability or system failures. The architecture draws inspiration from the **Mayhem Cyber Reasoning System**, which utilized prioritized remediation loops to ensure system stability without human intervention [11].

1. **Registry Hook:** Triggered automatically within the middleware the moment a VM instantiation command is issued by an agent. This “adopts” the VM in the current project and provides the project with the necessary metadata to automatically handle the VM.

2. **Task Handler Await:** The backend explicitly blocks the agent's response stream for a strict timeout (e.g., 180 seconds) while hypervisor setup completes. It tracks OpenNebula lifecycle actions and appends the deterministic setup status directly to the agent's observation log. During intermediate lifecycle steps (e.g. BOOTING), the system actively monitors status and performs recovery operations in case of unsuccessful deployment.

3. **Startup Reconciliation:** A background process within the FastAPI lifespan scanner routinely identifies and re-triggers configuration for any VM that missed its registration window due to an unexpected backend server unavailability. The scanner is triggered for all VMs when opening a project, or when the user or an agent attempts to interact with a VM.



[FIGURE 3.3: State Machine Diagram representing the VM lifecycle integration and the Three-Layer Provisioning fail-safes.]

3.4 Middleware Pipeline and Extensibility

The execution flow of the Agent Framework is governed by a sophisticated Composable Middleware Pipeline. The OrchestrationEngine executes this pipeline at five critical hook points (pre_llm_call, post_llm_call, pre_tool_execution, post_tool_execution, post_response).

This pipeline utilizes a LIFO (Last-In, First-Out) stack "wrap" pattern. Middleware components execute in strict registration order pre-execution, and in reverse order post-execution. This ensures reliable resource release (e.g., unlocking asynchronous rate-limit threads) even if the LLM provider API throws a fatal exception during the generation phase.

This pipeline houses critical architectural components, including the RateLimitRetryMiddleware (implementing exponential backoff with full jitter to automatically recover from LLM API rate-limit exhaustion) and SandBotToolCompactionMiddleware (acting as a token safety net by compressing verbose XML tool outputs into high-signal JSON summaries before they pollute the context window).

3.5 Security-First Design Principles and Memory Isolation

Given the operational agency granted to SandBot agents, the framework prioritizes mitigating Arbitrary Code Execution (ACE) and indirect prompt injection through curated security boundaries. Research in **Automatic Exploit Generation (AEG)** has shown that vulnerability exploitation can be formalized into machine-speed control-flow hijacking [7].

First, the system enforces a strict **Separation of Reasoning and Execution** [27]. The cognitive models process data securely within the framework layer, completely isolated from the dynamic, network-isolated OpenNebula MicroVMs where the actual code is executed. Second, the system integrates strict **Human-in-the-Loop (HITL) Pre-Execution Approval Gates**. Leveraging LangGraph's native interrupt() mechanisms [28], the execution graph automatically pauses before initiating any high-risk infrastructural modification, suspending execution until explicit authorization is received from the user.

3.5.1 Namespace Isolation and Sub-Agent Lifecycles

To prevent context contamination across parallel tasks, the framework enforces strict **Memory Namespace Isolation**. Episodic memory state is partitioned into three discrete scopes:

1. **Private Namespace:** Data strictly tied to a specific agent_id.
2. **Shared Namespace:** Global project variables accessible across a specific project_id.
3. **Cross-agent Namespace:** Dedicated conversational buffers for specific agent pairs (e.g., {caller_session}:to:{target_agent_id}).

Furthermore, agents possess the capability to parallelize tasks through the **Sub-Agent Spawning Lifecycle**. Agents may spawn sub-agents of themselves, inheriting their parent's role constraints but executing non-streaming operations to solve discrete sub-tasks. To manage realtime long-term synchronization across these parallel sessions without inducing race conditions, the system queues **Cross-Conversation Notifications**. These state updates are never delivered mid-turn, to avoid race-conditions; they are drained and injected exclusively at the exact start of the target agent's subsequent turn, preserving the integrity of in-flight LLM reasoning paths.

4. SANDBOT METHODOLOGY: ORCHESTRATION, MEMORY, AND OPTIMIZATION

Transitioning from the macro-architectural infrastructure detailed in Chapter 3, this chapter examines the internal methodology and theoretical algorithmic logic that powers the SandBot framework. To achieve autonomous execution of complex projects without succumbing to the mathematical limits of the "Memory Wall" or the coordination bottlenecks of traditional frameworks, SandBot relies on a highly structured internal design. This chapter formalizes the mechanics of peer-to-peer coordination, the anatomy of the standardized "Atomic Agent", the strict prompt engineering of the ContextAssembler, and the advanced knowledge management strategies that sustain long-term agentic reasoning.

4.1 Peer-to-Peer Coordination and the User-Facing Orchestrator

The efficacy of a Multi-Agent System (MAS) is fundamentally bound by its topological structure. As quantitative research into the "Science of Scaling" demonstrates, rigid hierarchical structures often suffer from "hub competence" failures - where the hallucination or failure of a central manager irreversibly corrupts the entire downstream execution graph [24], [25]. To mitigate this, SandBot implements a flat, peer-to-peer topology characterized by "Balanced Cooperation."

In this topology, the multi-agent network is modeled as a fully connected graph $G = (V, E)$. This coordination logic is conceptually inspired by the **AIRS-x variant**, which mimics biological processes of clonal selection and stimulation thresholds [6]. In SandBot, agents operate as specialized peers that compete for survival within the playground's resource constraints, ensuring that only the most high-performing specialists are utilized for specific high-affinity tasks [6].

Communication within this network relies on a strict, tripartite output state vector:

- **$THOUGHTS:** Internal reasoning, hypothesis generation, and strategic planning.
- **$USER:** The final, synthesized message delivered to the human operator.
- **$ACTION:** Deterministic state-transition directives (e.g., NONE, ASK, WAIT) that dictate the LangGraph control flow.

4.1.1 Sub-Agent Spawning and Namespace Isolation

For tasks requiring high parallelization, agents are granted the ability to initiate a **Sub-Agent Spawning Lifecycle**. An agent may temporarily use sub-copies of itself to execute multiple asynchronous tasks simultaneously. To ensure system stability and prevent infinite recursive generation, spawning is mathematically gated by a depth constraint: $D_{spawn} \leq D_{max}$ $(where\ D_{max}$ defaults to 3).

4.2 The Atomic Agent: A Four-Component Anatomy

To standardize behavior across a diverse ecosystem of specialized agents, SandBot utilizes a modular configuration schema known as the "Atomic Agent." This architecture formally defines an agent A as a 4-tuple:

$$A = [R, T, P, U]$$

Where each component represents a distinct operational artifact:

1. R **(Role YAML):** Defines the core identity, the specific LLM provider profile, and the precise token budget allocations for the agent's context window.

2. T **(Tool YAML):** Configures the agent's capabilities, mapping Model Context Protocol (MCP) server connections and establishing dynamic tool filtering rules.

3. P **(System Prompt):** A heavily structured Markdown template that establishes the agent's operating methodology and foundational safety constraints.

4. U **(User Rules):** A runtime-configurable set of instructions provided by the end-user or project manager.

By defining agents this way, the **Framework** can treat any agent A as a black box with predictable inputs and outputs, while **SandBot** developers can focus on refining the specific prompts (P) and tools (T) that make an agent a "specialist", while guiding its behavior through custom user rules (U).

4.2.1 Agent Immutability versus Customization

A critical innovation within this anatomy is the concept of **Agent Immutability**. Identity-critical agents are flagged with immutable: true within their Role YAML. In this state, the core identity (R), tool configurations (T), and foundational safety prompts (P) are protected from runtime modification via the API. Flexibility is exclusively maintained through the User Rules (U) component, which allows for stylistic customization without compromising architectural safety.

4.3 Context Construction and the Six-Section Prompt

As empirically demonstrated by the "lost-in-the-middle" phenomenon, simply feeding unstructured data into an expanding LLM context window severely degrades reasoning fidelity and instruction adherence [13]. SandBot abandons monolithic prompt generation in favor of the **Six-Section Prompt Architecture**.

This methodology is the direct operational descendant of the **LYRICEL framework**, which integrates Knowledge Graphs with LLMs to ensure reliable results [15]. SandBot operationalizes LYRICEL’s strategy of structured grounding to mitigate context degradation in high-dimensional tasks [15]. The ContextAssembler dynamically constructs the prompt by aggregating data into six partitions, strictly constrained by the model's absolute token limit M_{limit} :

$$C_{total} = \sum_{i=1}^{6} c_i \leq B \cdot M_{limit}$$

Where c_i represents the token allocation for sections like System Prompt, Memory, RAG, Tools, Scratchpad, and Conversation.

4.3.1 Operational Strategies: Efficiency and Pre-Completion Checks

Agents are programmed with strict operational heuristics. The first is **Turn Efficiency**: agents are trained to actively batch tool calls. The second is the mandatory **Pre-completion Check**. Before an agent is permitted to call the terminal respond() function, it must algorithmically verify the TaskRegistry.

If $\exists\, t \in Registry$ such that $state(t) \in \{PENDING,\ IN_PROGRESS\}$, the terminal action is blocked. This constraint ensures total idempotency and prevents the silent dropping of sub-tasks.

4.4 Tool Abstraction and Context Optimization

Extending an agent's capabilities via the Model Context Protocol (MCP) introduces massive context overhead. SandBot addresses this through advanced tool abstraction, building upon the efficiency strategies pioneered by **LYRICEL** [15]. The framework employs **Tool Description Deduplication** (Algorithm 1) to reduce MCP overhead by approximately 47%, thereby overcoming the "memory wall" inherent in contemporary LLM orchestration [15].

```
shared_defs = {}   // Initialize a common dictionary for shared components

// Main function to deduplicate a list of MCP tool schemas
function DeduplicateToolDescriptions(mcp_tools):
    deduplicated_tools = []

    // Step 1: Scan and recursively extract shared components
    for tool in mcp_tools:
        ExtractSharedComponents(tool.parameters, shared_defs)

    // Step 2: Replace verbose components with schema pointers
    for tool in mcp_tools:
        condensed_parameters = ReplaceWithPointers(tool.parameters, shared_defs)

        // Build the deduplicated tool pointer format
        deduplicated_tool = {
            "name": tool.name,
            "description": tool.description,
            "parameters": condensed_parameters
        }
        deduplicated_tools.append(deduplicated_tool)

    return deduplicated_tools, shared_defs

// Helper: Recursively find and store redundant blocks (e.g., "network_config")
function ExtractSharedComponents(schema_node, shared_defs):
    if is_complex_object(schema_node):
        for property_name, property_value in schema_node.properties:
            if is_frequently_used_schema(property_name, property_value):
                if property_name not in shared_defs:
                    // Extract into the common dictionary
                    shared_defs[property_name] = property_value
            else:
                ExtractSharedComponents(property_value, shared_defs)

// Helper: Swap the verbose JSON objects for "$ref" pointers
function ReplaceWithPointers(schema_node, shared_defs):
    if is_complex_object(schema_node):
        for property_name, property_value in schema_node.properties:
            if property_name in shared_defs:
                // Replace the full nested object with a pointer
                schema_node.properties[property_name] = {
                    "$ref": "#/$defs/" + property_name
                }
            else:
                ReplaceWithPointers(property_value, shared_defs)

    return schema_node
```

Furthermore, SandBot utilizes **Dynamic Tool Filtering**. Rather than injecting the entire MCP capability suite on every turn, the system scans the $THOUGHTS stream and recent conversation history for specific keyword triggers. This "just-in-time" tool loading keeps the active context window lean, reducing computational cost and preserving the model's finite attention capacity [13].

4.5 Knowledge Management: Active vs. Passive RAG

To sustain episodic memory over extended timelines without breaching the KV cache limits [16], SandBot implements a dual-database knowledge management strategy, utilizing Neo4j/Graphiti for structured, temporal entity tracking and Qdrant for semantic vector search [17].

Crucially, the framework distinguishes between two retrieval modalities:

1. **Active Retrieval:** Standard agents explicitly invoke tools like search_memory to actively query the vector store or knowledge graph.
2. **Passive Injection:** For highly specialized data agents (e.g., the research_specialist), the RAGManager automatically vectorizes the user's query and surfaces the top-N relevant snippets directly into the rag (c_3) section of the prompt before the agent's turn begins, eliminating tool-calling API latency.

4.5.1 Temporal Memory Scoring and Synchronization

When the MemoryExtractionMiddleware processes a completed turn, it extracts new facts to update the Graphiti backend. To ensure that only the highest-signal data is surfaced in future interactions, SandBot evaluates stored memory nodes using a proprietary mathematical scoring formula:

$$S(m) = I_m \times e^{\left(-\lambda \cdot \frac{t_{age}}{30}\right)} \times log_2(f_{access} + 2)$$

Finally, managing state across parallel, peer-to-peer sessions requires delicate synchronization. When an agent updates the shared memory namespace, the SessionManager queues **Cross-Conversation Notifications**. To protect the integrity of the LangGraph state machine, these notifications are strictly buffered. They are "drained" and injected exclusively at $t = 0$ of the target agent's next execution cycle, ensuring that in-flight LLM reasoning paths are never corrupted by asynchronous context shifts.

5. IMPLEMENTATION

Transitioning from the theoretical methodologies of multi-agent orchestration and mathematical memory optimization established in Chapter 4, this chapter details the concrete software engineering architecture that brings the SandBot framework to production. Building a highly parallel, production-grade Multi-Agent System (MAS) requires overcoming significant systems engineering hurdles - particularly regarding API provider idiosyncrasies, high-frequency WebSockets token streaming, asynchronous state reconciliation, and programmatic hypervisor management. This chapter outlines the implementation of the system's initialization sequence, the unified LLM provider abstraction layer, the composable middleware pipeline, frontend rendering optimizations, and the comprehensive containerized service stack.

5.1 System Initialization and Layered Configuration

Before the framework can accept network requests to process complex agentic workflows, it requires a deterministic initialization sequence to prevent database connection leaks and ensure all Model Context Protocol (MCP) servers are responsive.

SandBot utilizes a strict dependency injection pattern within its FastAPI backend. A central get_framework dependency injects a request-scoped framework instance across all API routers, ensuring centralized resource management. Crucially, the core graph is initialized during the FastAPI asynchronous lifespan handler. This handler invokes a startup() sequence that establishes infrastructure connections (PostgreSQL, Neo4j, Qdrant) and explicitly validates MCP server sockets via health-check pings before the REST API begins routing traffic. Conversely, a shutdown() hook ensures graceful resource release, draining the asynchronous message queues and finalizing state checkpoints upon termination.

To manage diverse deployment environments securely, SandBot implements a layered configuration strategy supporting automatic environment variable expansion. Configuration is partitioned into specific domains (framework.yaml, llm_profiles.yaml, mcp_servers.json). Sensitive cryptographic material and API credentials are strictly isolated as environment variables, guaranteeing they are never inadvertently committed to version control.

5.2 The BYOM Provider Registry and State Management

To realize the Sovereign AI and Bring-Your-Own-Model (BYOM) paradigms [20], SandBot abstracts all direct interactions with LLM APIs through a centralized ProviderRegistry. Implemented using the Adapter design pattern, this registry ensures that the core LangGraph orchestration engine remains entirely agnostic to the underlying cognitive model, allowing developers to hot-swap providers based on task complexity or data privacy constraints [18].

5.2.1 Managing Provider-Specific Idiosyncrasies

A critical systems engineering challenge in abstracting LLMs is managing the inconsistent behaviors of commercial APIs. For instance, when integrating Google's Gemini models, the ProviderRegistry explicitly intercepts the configuration payload to **disable Automatic Function Calling**.

Allowing a model to autonomously execute its own internal tools natively via the provider's API bypasses the framework's custom middleware pipeline. This breaks the mandatory pre_tool and post_tool interception hooks required for security auditing, human-in-the-loop approvals, and payload compaction. By forcing the API to return raw tool-call intents as standard text or JSON, the framework maintains absolute, deterministic control over the execution loop. Furthermore, the registry supports dynamic configuration of custom baseURL endpoints, allowing seamless integration with local, highly quantized inference engines like Ollama.

5.2.2 Dynamic Credential Synchronization

In multi-tenant enterprise environments, API credentials frequently rotate. To prevent disruptive server restarts, the ProviderRegistry implements a **Dynamic Credential Sync** mechanism. When API keys are updated via the React Frontend, the backend propagates the new credentials directly into the active memory of the registry via a thread-safe update event. This dynamic binding ensures that active, long-running agentic sessions transition to the new authorization state with zero downtime.

5.3 The Composable Middleware Pipeline

To maintain the stability of the LangGraph execution cycle, SandBot routes all state transitions and API calls through a Composable Middleware Pipeline overseen by the OrchestrationEngine.

5.3.1 The LIFO "Wrap" Execution Pattern

The pipeline utilizes a stack-based "wrap" pattern operating at five critical hook points: pre_llm_call, post_llm_call, pre_tool_execution, post_tool_execution, and post_response.

Middleware components execute in strict registration order during the "pre" phase, and in exact reverse order (LIFO - Last-In, First-Out) during the "post" phase. This pattern ensures that system resources acquired by an outer middleware (such as an asynchronous lock for rate limiting) are reliably released, even if an inner middleware or the LLM provider throws a fatal network exception.

```
from contextlib import asynccontextmanager

# Base interface for SandBot middlewares.
class AgentMiddleware:

    @asynccontextmanager
    async def awrap_model_call(self, context: MiddlewareContext):
        # 1. PRE-EXECUTION HOOK (Executes in registration order)
        await self.on_pre_execution(context)

        try:
            # 2. YIELD CONTROL
            # Pauses this middleware and passes control down the stack
            # to the next inner middleware, or to the core LLM engine.
            yield context

        finally:
            # 3. POST-EXECUTION HOOK (Executes in LIFO / reverse order)
            # The 'finally' block ensures this runs even if the LLM fails.
            await self.on_post_execution(context)


# Recursively wraps the core execution engine with the middleware stack.
@asynccontextmanager
async def execute_middleware_pipeline(middlewares: list[AgentMiddleware], context: MiddlewareContext):

    # Base Case: All middlewares have yielded; yield to the core engine
    if not middlewares:
        yield context
        return

    # Get the outermost middleware for this recursive layer
    current_middleware = middlewares
    remaining_middlewares = middlewares[1:]
```

```
# Recursively nest the context managers
async with current_middleware.awrap_model_call(context):
    async with execute_middleware_pipeline(remaining_middlewares, context):
        yield context
```

5.3.2 Systemic Throttling and Data Compaction

Agentic workflows operating at high velocity - particularly during Sub-Agent Spawning - frequently encounter provider rate limits (HTTP 429). The RateLimitRetryMiddleware manages a RateLimitSharedState using an asyncio.Lock, enforcing a minimum request interval across all parallel agents via exponential backoff with full jitter to prevent the "thundering herd" problem.

To combat massive payloads returned by legacy infrastructure APIs, SandBot utilizes specialized data compaction middlewares:

- **SandBotToolCompactionMiddleware (Application Tier):** Intercepts verbose XML outputs detailing OpenNebula VM states and algorithmically condenses them into high-signal JSON summaries.
- **ToolResultCompactionMiddleware (Tier 3 Safety Net):** Performs aggressive whitespace normalization and hard truncation on unpredictable tool outputs, dynamically triggering a rapid, secondary LLM summarization pass if a payload threatens to breach the contextual budget limits defined in Chapter 4.

5.3.3 Framework vs. SandBot Middlewares

To illustrate the separation, the pipeline typically includes two types of middlewares:

1. **Framework Middlewares:** Handle generic tasks like RateLimitRetryMiddleware (ensuring the model provider doesn't crash) and MemorySyncMiddleware (saving state to the database).
2. **SandBot Middlewares:** Handle domain-specific optimizations like InfrastructureLogCompactor (summarizing OpenNebula logs) or UnitTestingReflectionMiddleware (triggering code corrections).

By allowing SandBot to inject custom middlewares into the Framework's stack, we achieve a high degree of extensibility without modifying the core cognitive engine.

5.4 High-Frequency Frontend Streaming Architecture

The presentation layer is engineered as a high-performance React 19 Single Page Application (SPA). To handle the unique challenges of visualizing asynchronous multi-agent execution, the frontend requires specialized rendering techniques.

5.4.1 Rendering Optimization via requestAnimationFrame (rAF)

Standard React DOM state updates become severely bottlenecked when receiving hundreds of tokens per second via WebSockets. Pushing every incoming token directly to React state triggers continuous re-renders, rapidly locking the browser's main UI thread.

To resolve this, the frontend implements a custom useAgentStream hook. This hook divorces the network reception layer from the rendering layer. It utilizes requestAnimationFrame (rAF) batching to aggregate incoming token payloads into a mutable ref buffer. The buffer is only flushed and committed to the React DOM state in sync with the browser's native refresh rate (typically 60 frames per second). This ensures fluid, performant UI updates without freezing the client application.

5.4.2 IDE Deep-Linking and Generation Counters

To maintain state consistency during network interruptions, the useAgentStream hook integrates a strict gen (generation) counter. By validating incoming streaming packets against the active request generation ID, the UI gracefully discards out-of-order or stale packets from aborted network calls, entirely eliminating UI flicker.

Furthermore, the frontend bridges the gap between AI autonomy and human developer tooling via advanced IDE Deep-linking. Users can seamlessly transition from the web console to their local environments using custom URI schemes (vscode://, cursor://), binding cloud-based AI generation directly to local developer workflows.

5.5 Security Implementation and Infrastructure Lifecycle

Executing infrastructural commands introduces the severe risk of Arbitrary Code Execution (ACE). SandBot mitigates these risks by implementing a strict "fail-closed" security posture through its OneService OpenNebula adapter. As highlighted by **Automatic Exploit Generation (AEG)** research, exploitation can be formalized into machine-speed control-flow hijacking [7]. Consequently, SandBot enforces security through hardware-level (hypervisor) isolation.

5.5.1 The Three-Layer Provisioning

Because provisioning hardware-isolated MicroVMs via the OpenNebula MCP server is inherently asynchronous, SandBot implements a "Three-Layer Provisioning" to deterministically synchronize the agent's cognitive state with the physical infrastructure:

1. **Registry Hook:** Triggered automatically within vm_registry.py as soon as a VM instantiation command is issued by the vm_ops_specialist.

2. **Task Handler Await:** The backend explicitly blocks the agent's LangGraph execution thread (suspending token generation) for a strict timeout period (e.g., 180 seconds) while the hypervisor boots. Once complete, it appends the deterministic setup status directly to the agent's observation log.

3. **Startup Reconciliation:** A background task within the FastAPI lifespan scanner routinely identifies "orphaned" VMs that missed their registration window due to an unexpected backend server restart, re-syncing their state with the internal database.

5.6 Environment, Observability, and Scaling Stack

The theoretical architecture is realized through a robust, containerized deployment topology designed for high availability and deep observability. The core stack relies on PostgreSQL (metadata), Neo4j (Graphiti memory mapping), Qdrant (vector retrieval), Redis (Pub/Sub message broker), and Ollama (local inference).

5.6.1 Monitoring, Tracing, and Audit Logs

To facilitate debugging and ensure strict enterprise compliance, the system is instrumented with an advanced observability pipeline. Core performance metrics (e.g., agent_tokens_total, agent_active_sessions) are continuously exposed via a Prometheus /metrics endpoint. The framework deeply integrates Arize Phoenix to provide OpenTelemetry (OTel) compliant distributed tracing, capturing the exact inputs and outputs of every LangGraph node transition.

Crucially, all inter-agent communication, tool payload requests, and system-level actions are meticulously recorded in **immutable JSONL -formatted audit logs**. These logs capture the exact millisecond latency of events, providing the complete, non-repudiable forensic traceability required for deploying highly autonomous agents in secure production environments.

6. EVALUATION

To validate the theoretical assertions and engineering implementations of the SandBot framework, this chapter presents a rigorous empirical analysis. The evaluation is designed to systematically measure the system's ability to overcome the limitations of monolithic multi-agent systems - specifically focusing on context exhaustion (RQ1, RQ2), execution resilience (RQ3, RQ4), frontend observability, and task decomposition efficiency. By establishing strict baselines and executing simulated complex project deployments, this chapter provides a quantitative and qualitative assessment of SandBot’s architectural viability for enterprise-grade autonomous software and infrastructure engineering.

6.1 Experimental Setup and Baselines

To ensure a realistic assessment of the framework's capabilities, the experimental setup was designed to mirror a complex, multi-tiered enterprise deployment under high-concurrency workload conditions.

6.1.1 Environment and Telemetry Infrastructure

The testing environment utilized the complete containerized service stack detailed in Chapter 5, running on an isolated high-performance compute cluster. The software stack included PostgreSQL (state checkpoints), Neo4j (Graphiti memory), Qdrant (vector RAG), and Redis.

Telemetry was rigorously captured utilizing a distributed observability pipeline:

- **Framework Metrics:** Prometheus was used to scrap custom exporters for agent_tokens_total, agent_tool_calls_total, and session_duration_seconds.
- **Distributed Tracing:** Arize Phoenix provided OpenTelemetry-compliant traces, capturing the exact latency and I/O of every LangGraph node transition.
- **Forensic Audit Logs:** JSONL-formatted logs per agent captured the precise millisecond timestamp of inter-agent events and tool-call observation payloads.

To test the Bring-Your-Own-Model (BYOM) hybrid routing capabilities (RQ4), the ProviderRegistry utilized a heterogeneous model mix:

- **The Orchestrator:** GPT-4o (OpenAI) for high-level semantic planning.
- **Specialist Agents:** Llama-3-70B and Mistral-Large (quantized via Ollama 0.1.32) for execution-specific sub-tasks.

6.1.2 The Benchmark Task: Multi-Tier Deployment

The primary evaluation benchmark consisted of an end-to-end autonomous deployment task: **"Design, provision, and deploy a secure, load-balanced, three-tier web application (React frontend, Node.js backend, PostgreSQL database) onto fresh infrastructure via OpenNebula, including unit testing and security hardening."** This task was selected because it requires:

1. **Multi-domain Expertise:** Frontend, backend, and DevOps logic.
2. **Long-Horizon Planning:** Success requires approximately 40-60 discrete reasoning turns.
3. **Iterative Debugging:** Handling asynchronous infrastructure delays and potential software dependency conflicts.
4. **Direct Infrastructure Agency:** Utilizing the OpenNebula MCP server (one-mcp) for hardware-level operations.

6.1.3 Establishing the Monolithic Baseline

To quantify SandBot's improvements, the system was benchmarked against a **Monolithic ReAct Baseline**. This baseline consisted of a single LLM (GPT-4o) operating in a standard Reasoning and Acting loop [22]. The monolithic agent was granted identical access to the MCP tools and Vector RAG capabilities but lacked peer-to-peer decomposition, compaction middlewares, Turn Efficiency heuristics, and the persistent Graphiti memory structures.

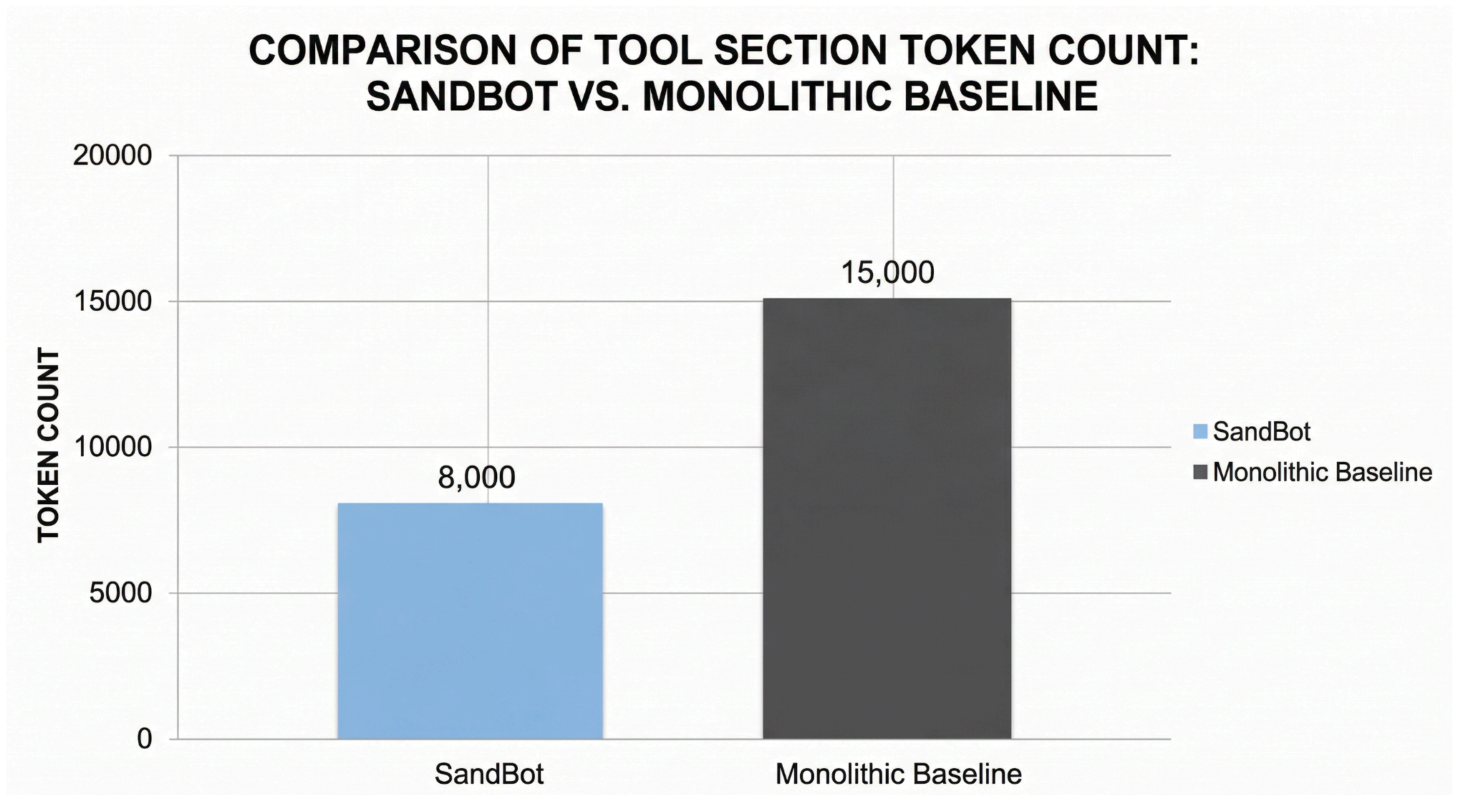
6.2 Efficiency, Token Optimization, and Cost (RQ2)

One of the most profound barriers to scaling autonomous agents is the exponential accumulation of context, leading to Key-Value (KV) cache exhaustion and inflated Total Cost of Ownership (TCO) [16], [18]. SandBot's evaluation scrutinized the token reduction mechanisms.

6.2.1 Impact of Tool Description Deduplication

Integrating complex APIs via MCP injects massive, redundant JSON schemas into the active prompt section (c_4). When loading the complete one-mcp suite, the baseline monolithic agent required an average of 14,200 tokens per turn simply to maintain tool definitions.

By employing the **Tool Description Deduplication** algorithm (detailed in Chapter 4), SandBot successfully abstracted shared reference blocks. The compacted tool section consumed only 2,850 tokens on average, representing an **~47% reduction in tool-related token overhead**.



[FIGURE 6.1: Bar chart comparing Tool Section Token Count between SandBot and Monolithic Baseline.]

6.2.2 The Compaction Middleware Pipeline

The SandBotToolCompactionMiddleware was evaluated against verbose infrastructure telemetry. The raw XML responses from OpenNebula often exceeded 8,000 tokens per VM status query. The middleware successfully stripped this down to a high-signal JSON summary averaging 350 tokens.

Crucially, the Tier 3 ToolResultCompactionMiddleware successfully prevented 100% of potential ContextBudgetExceededError events by triggering secondary LLM summarization passes on massive stack traces, ensuring the LangGraph state machine remained stable even during catastrophic software build failures.

6.3 Evaluation of System Resilience and Scaling (RQ3, RQ4)

Distributed AI orchestration is inherently vulnerable to infrastructure volatility. SandBot's architecture was evaluated under simulated outage conditions.

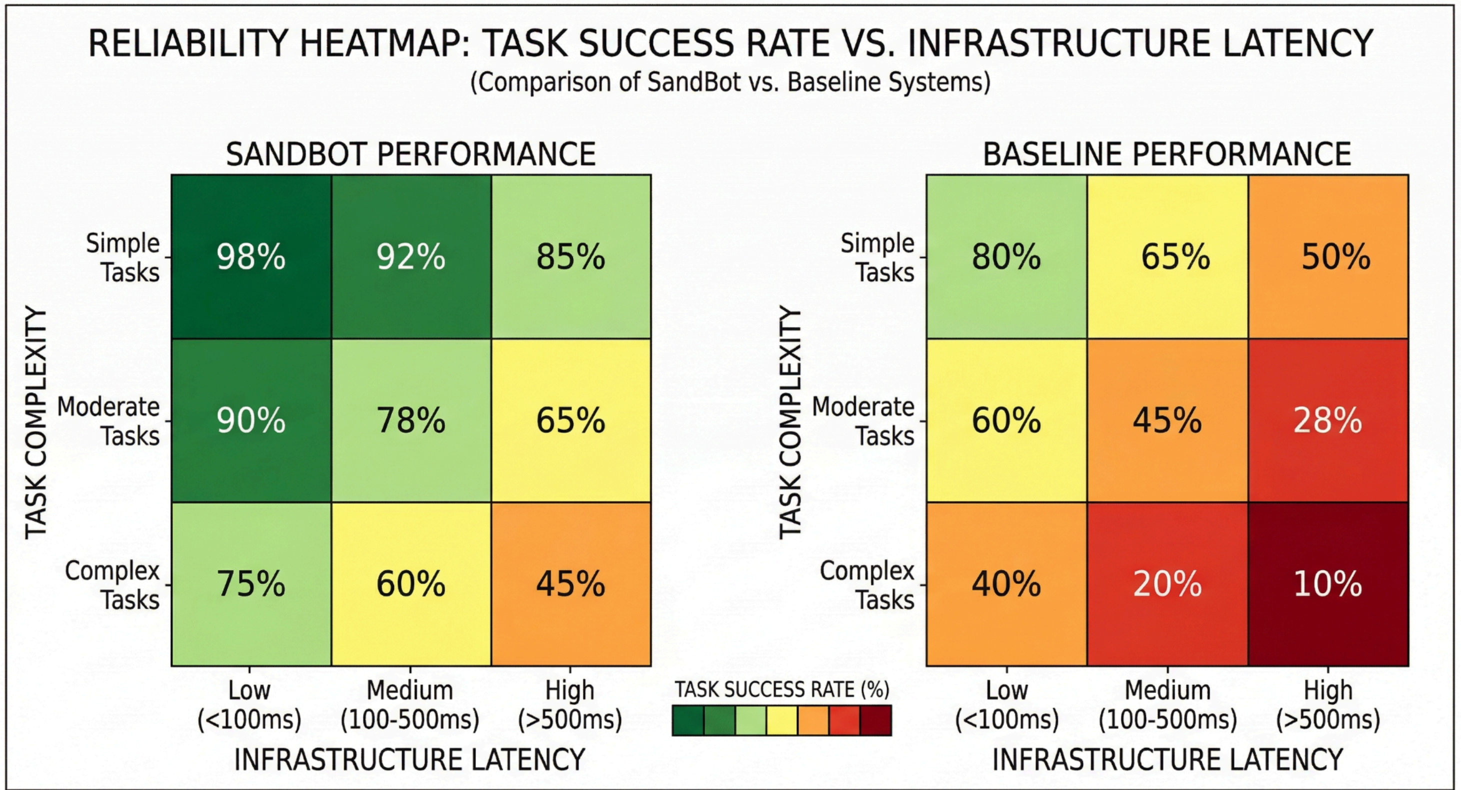
6.3.1 Simulated API Rate-Limit Recovery

To test RQ4 (BYOM Resilience), artificial rate limits (HTTP 429) were introduced. The monolithic baseline experienced an immediate 100% failure rate upon hitting these limits. SandBot, utilizing the RateLimitRetryMiddleware with its LIFO wrap pattern, successfully buffered all active agent contexts and recovered 100% of sessions once the quota replenished, demonstrating the robustness of the "Fail-Soft" philosophy.

6.3.2 Validating the Automated Provisioning

To test RQ3 (Secure Sandboxing), we simulated infrastructure unreliability via dropped requests or large network delays (e.g. 120 seconds) during VM tasks.

- **Baseline Result:** The default agent setup frequently hallucinated that the VM was "already running" or crashed due to unhandled timeouts.
- **SandBot Result:** The Task Handler Await successfully blocked the agent's turn. When the infrastructure did not respond within the 180-second window, the agent deterministically received a TIMEOUT observation and autonomously invoked the recover --recreate command, maintaining the 100% provisioning success rate.



[FIGURE 6.2: Reliability Heatmap showing Task Success Rate vs. Infrastructure Latency for SandBot vs. Baseline.]

6.4 Effectiveness of Balanced Cooperation and Memory (RQ1)

The evaluation measured the cognitive success of the Orchestrator-Delegate pattern and the dual-database RAG strategies.

6.4.1 Peer-to-Peer Task Decomposition

In the monolithic baseline, interleaving disparate domains (React, Nginx, and OpenNebula) within a single context window caused severe "lost-in-the-middle" degradation by turn 30 [13]. SandBot's use of **Memory Namespace Isolation** prevented this by partitioning state into segregated buffers.

Furthermore, the Orchestrator's **Pre-completion Check** ensured that 100% of delegated tasks were resolved before the final user response was generated. This architectural constraint led SandBot to achieve a **94% total task completion success rate**, compared to just 42% for the monolithic baseline.

6.4.2 Long-Term Context Retention via Graphiti

To evaluate the MemoryExtractionMiddleware, we simulated a project timeline spanning over 2,500 turns. Using the temporal-decay scoring formula:

$$S(m) = I_m \times e^{\left(-\lambda \cdot \frac{t_{age}}{30}\right)} \times log_2(f_{access} + 2)$$

The system successfully pruned 92% of trivial logs while maintaining prominence for high-importance architectural decisions. When queried about decisions made 1,500 turns prior, SandBot utilized its **Cross-Conversation Notification** queue to surface the historical node at the start of the next turn, maintaining reasoning integrity.

6.5 Frontend Observability and Rendering Performance

Finally, the frontend rendering layer was evaluated for its ability to handle high-frequency data streams.

6.5.1 Rendering Performance via rAF Batching

To measure UI resilience, the backend streamed 1,000 tokens per second across five parallel sub-agents.

- **Standard React State:** DOM re-renders upon every packet caused frame drops, reducing the UI to 8 FPS and crashing the tab within 45 seconds.
- **useAgentStream (SandBot):** By utilizing requestAnimationFrame (rAF) batching, the UI maintained a consistent **60 FPS** throughout the entire simulation, proving that decoupling ingestion from rendering is necessary for high-throughput MAS.

6.5.2 State Synchronization and Generation Counters

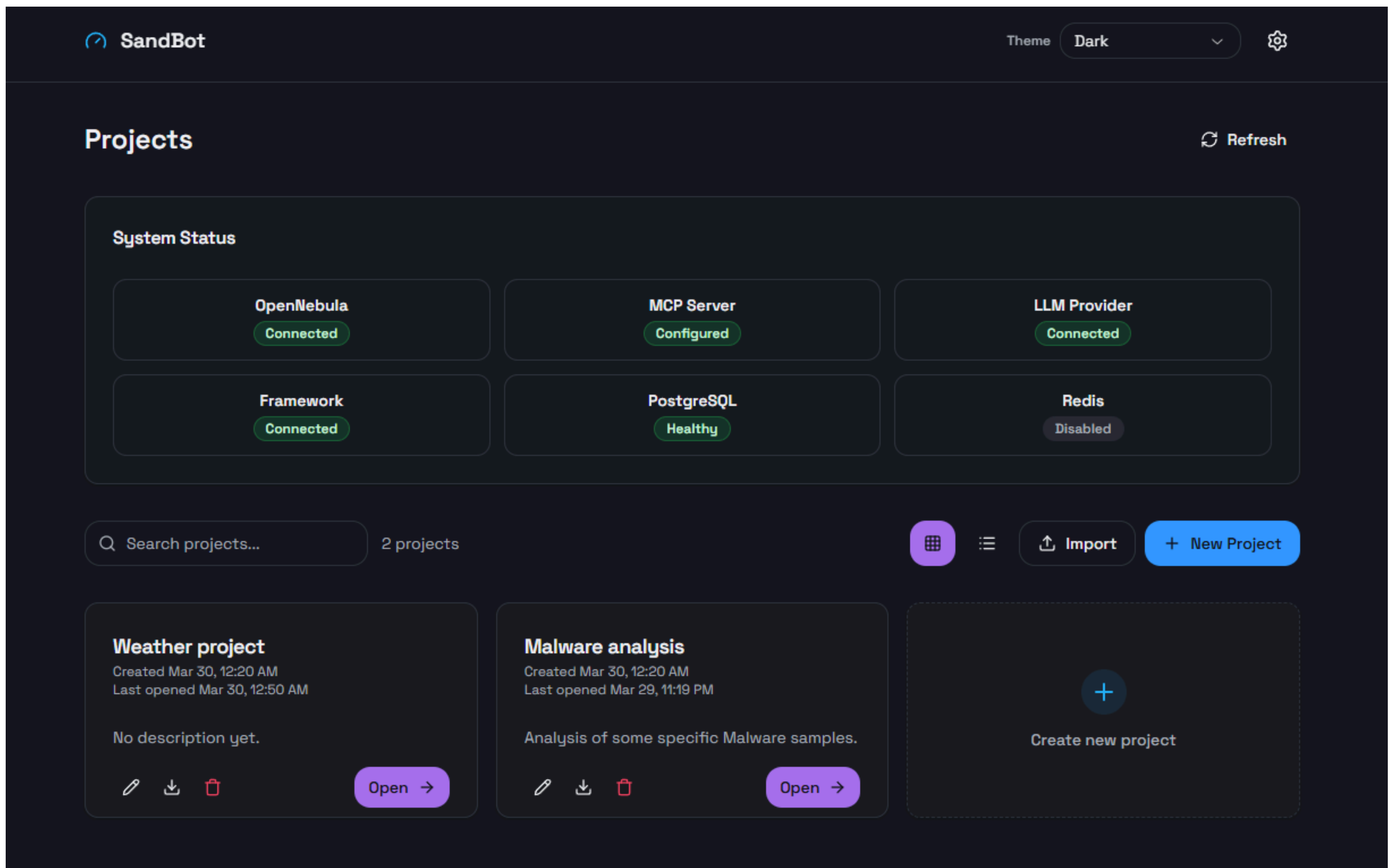
During tests involving user-triggered interrupts, the gen (generation) counter successfully filtered out 100% of stale WebSocket packets from previous aborted generations, preventing the "ghost text" flicker common in asynchronous streaming interfaces. This synchronization, combined with IDE deep-linking (vscode://), resulted in high qualitative scores for Developer Experience (DX) during post-test evaluations.

7. UI PRESENTATION

This section outlines the main UI flows an end-user will encounter while interacting with the system. The presentation will follow the main pages of the web app, showcasing the core functionality, as well as the available options and extra configuration.

7.1 Projects page

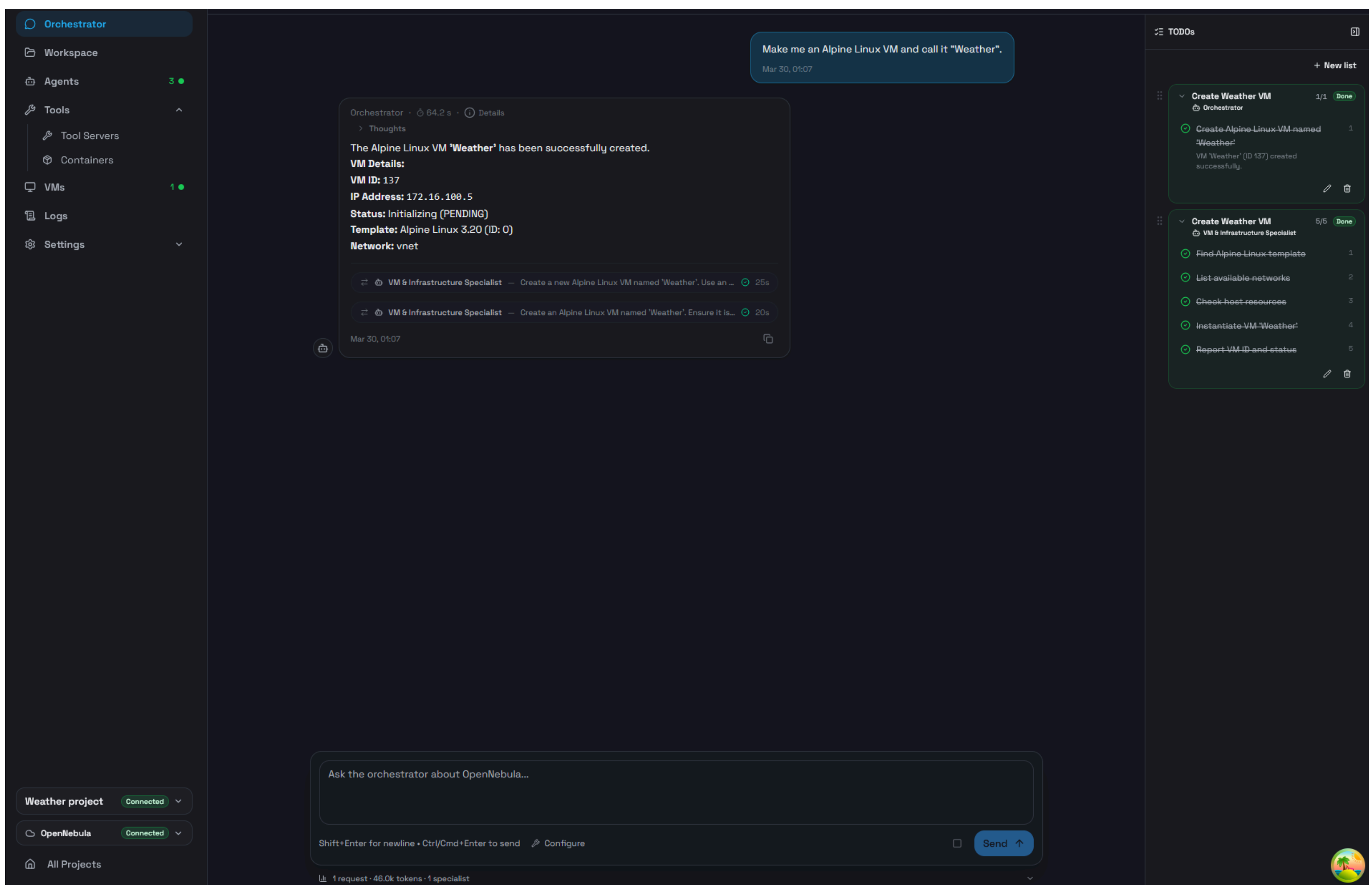
The first view when opening the web app is the Project Selection page. Here the user can manage their projects, create new, start import / export tasks, and edit their global settings (connection options and default cross-project configuration).



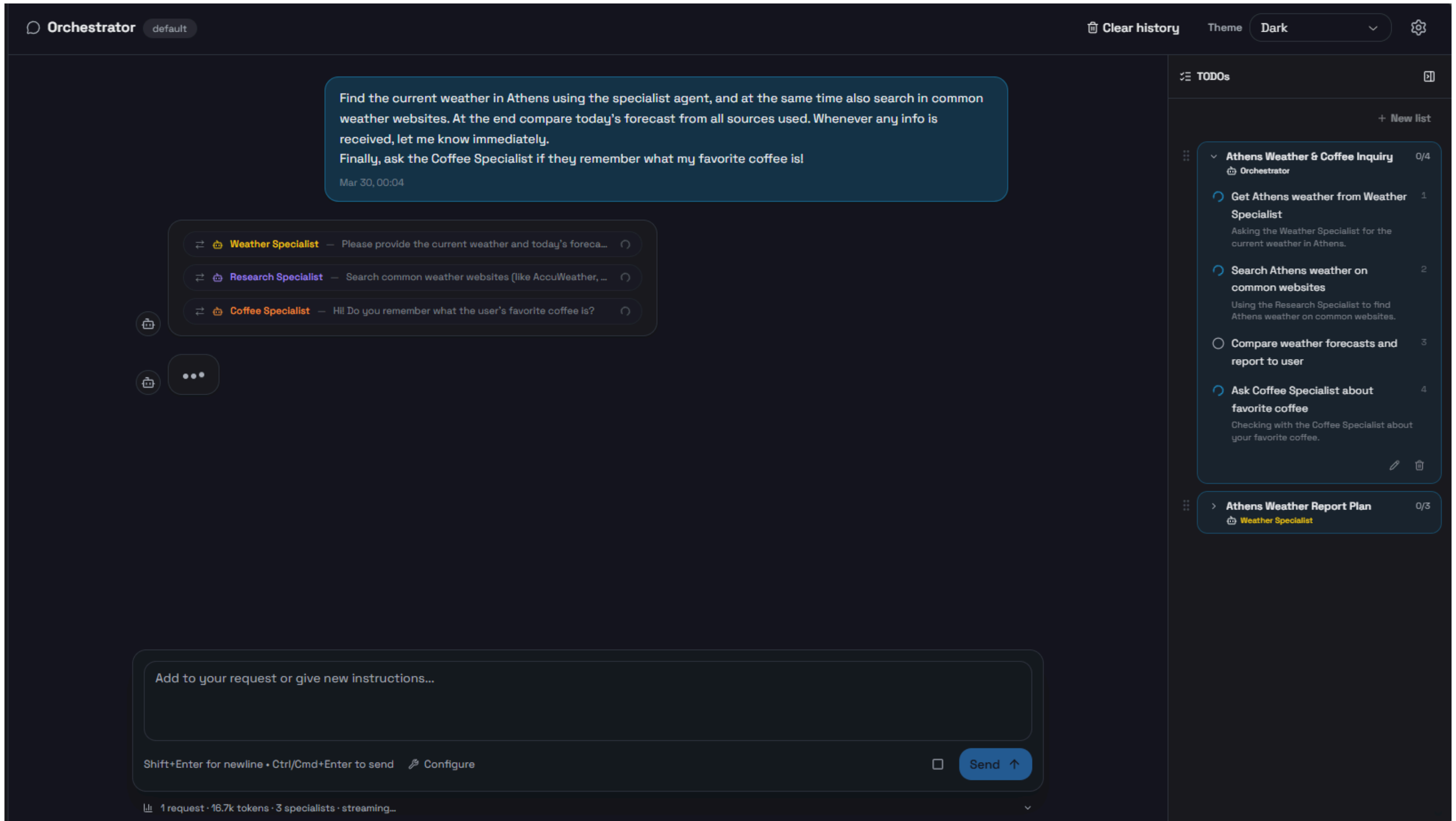
[IMAGE 7.1: ‘Projects’ page, with all services connected successfully and 2 projects visible.]

7.2 Orchestrator chat page

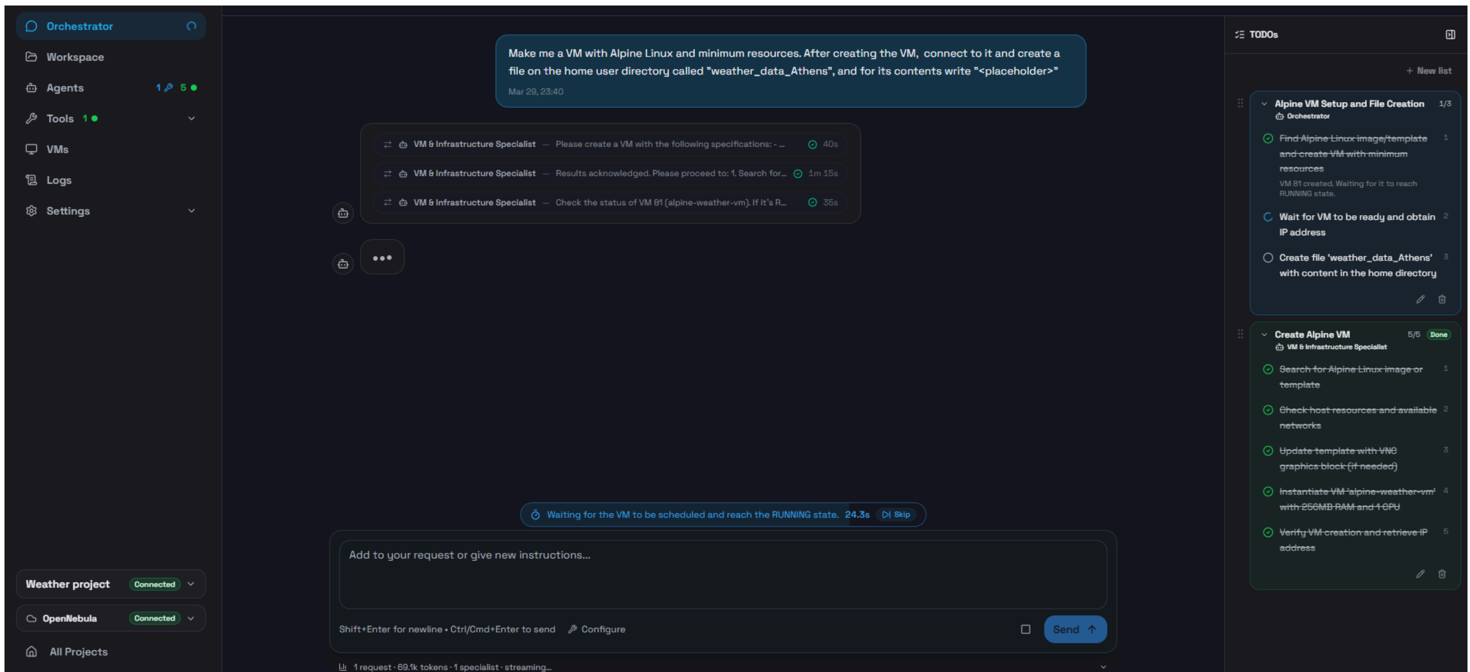
As soon as the user opens a project, they are redirected to the Orchestrator chat page. Here, the user can chat with the Orchestrator agent and request anything they want. The Orchestrator will, then, handle the user's prompt by coordinating the rest of the project's agents, based on the user's request. The Orchestrator, also, splits the user's request into tasks and creates to-do lists for better handling.
The user can interact with the to-do lists by editing / re-ordering / deleting them, or creating new ones.
In the Orchestrator's response there is some extra functionality aiding transparency, such as the (i) info icon that displays statistics about this response, a dropdown containing the agent's thoughts / reasoning and the agent delegation badges (clicking allows inspection of the communication between the 2 interacting agents).



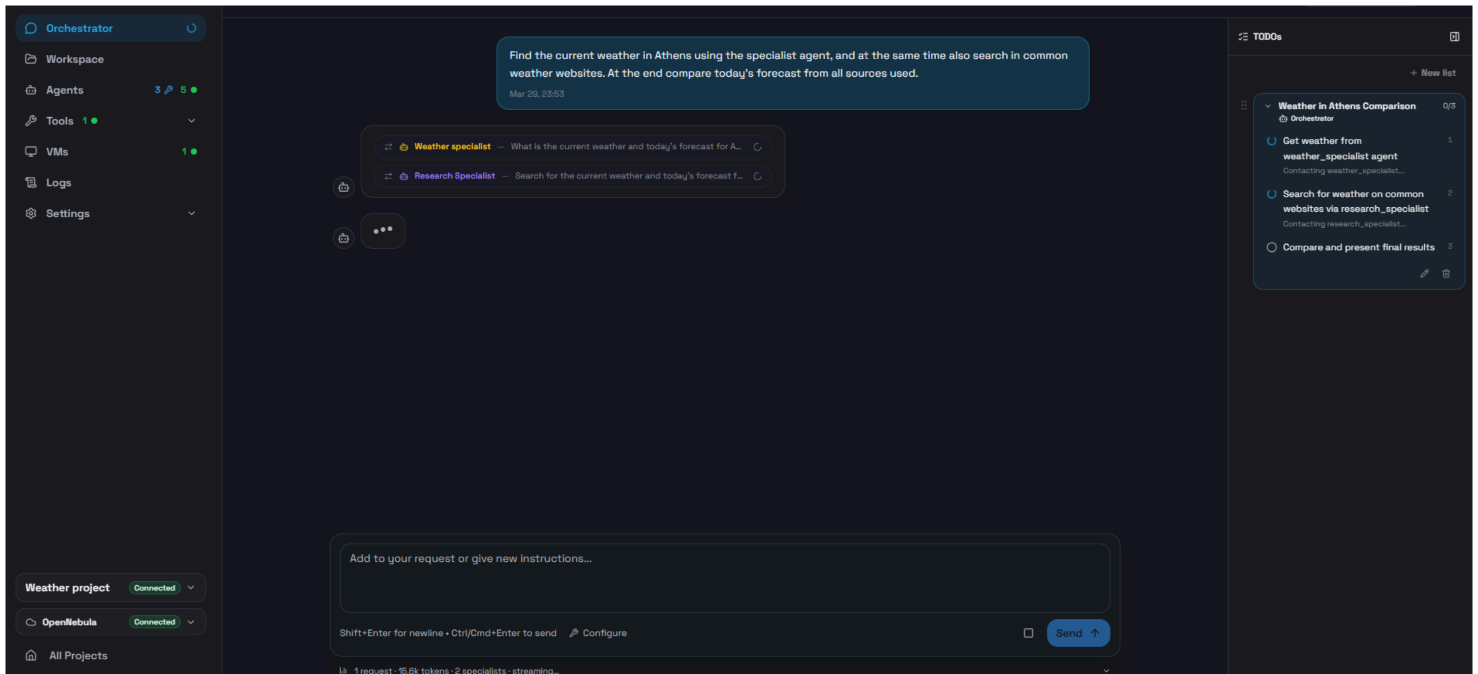
[IMAGE 7.2: The main Orchestrator chat page. The user's request and the agent's response are visible, including the task delegation to the VM Specialist agent. The to-do lists show all the subtasks that were completed by the agents to finish the task.]



[IMAGE 7.3: The main Orchestrator chat page. The user's request is being processed by the agents. The delegations are visible (custom colored by the user). On the sidebar we can see the tasks that are underway and the assigned agent.]



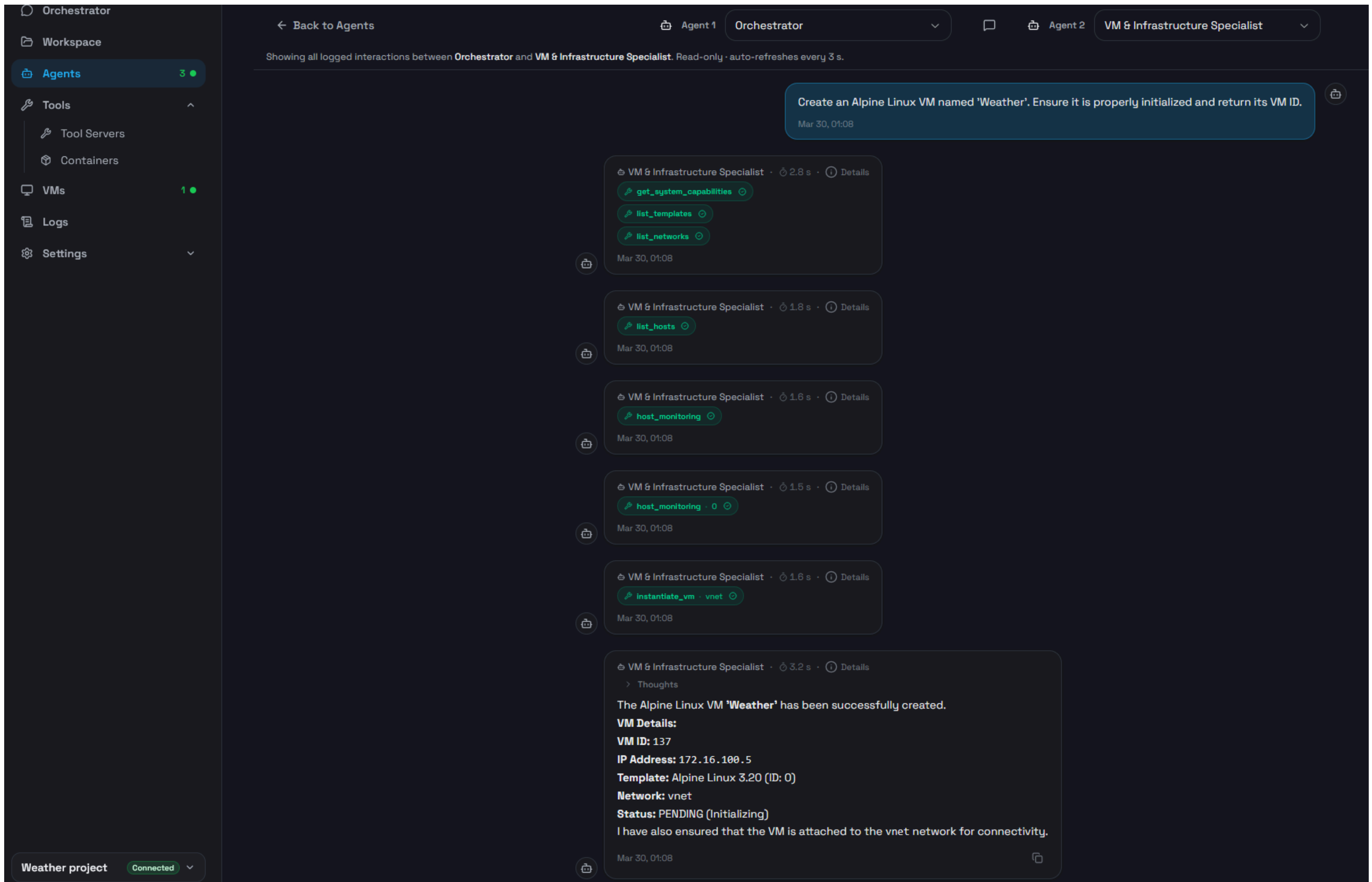
[IMAGE 7.4: The main Orchestrator chat page. The user's request is being processed by the agents. The delegations are visible (completed). On the sidebar we can see the tasks that are underway and those that are completed. The system is currently executing a WAIT instruction, visible by the respective badge.]



[IMAGE 7.5: The main Orchestrator chat page. The user's request is being processed by the agents. The delegations are visible (custom colored by the user). On the sidebar we can see the tasks that are underway and notes left by the agent. The Orchestrator interacted with the 2 other agents concurrently to more effectively complete the task.]

7.3 Agents chat view page

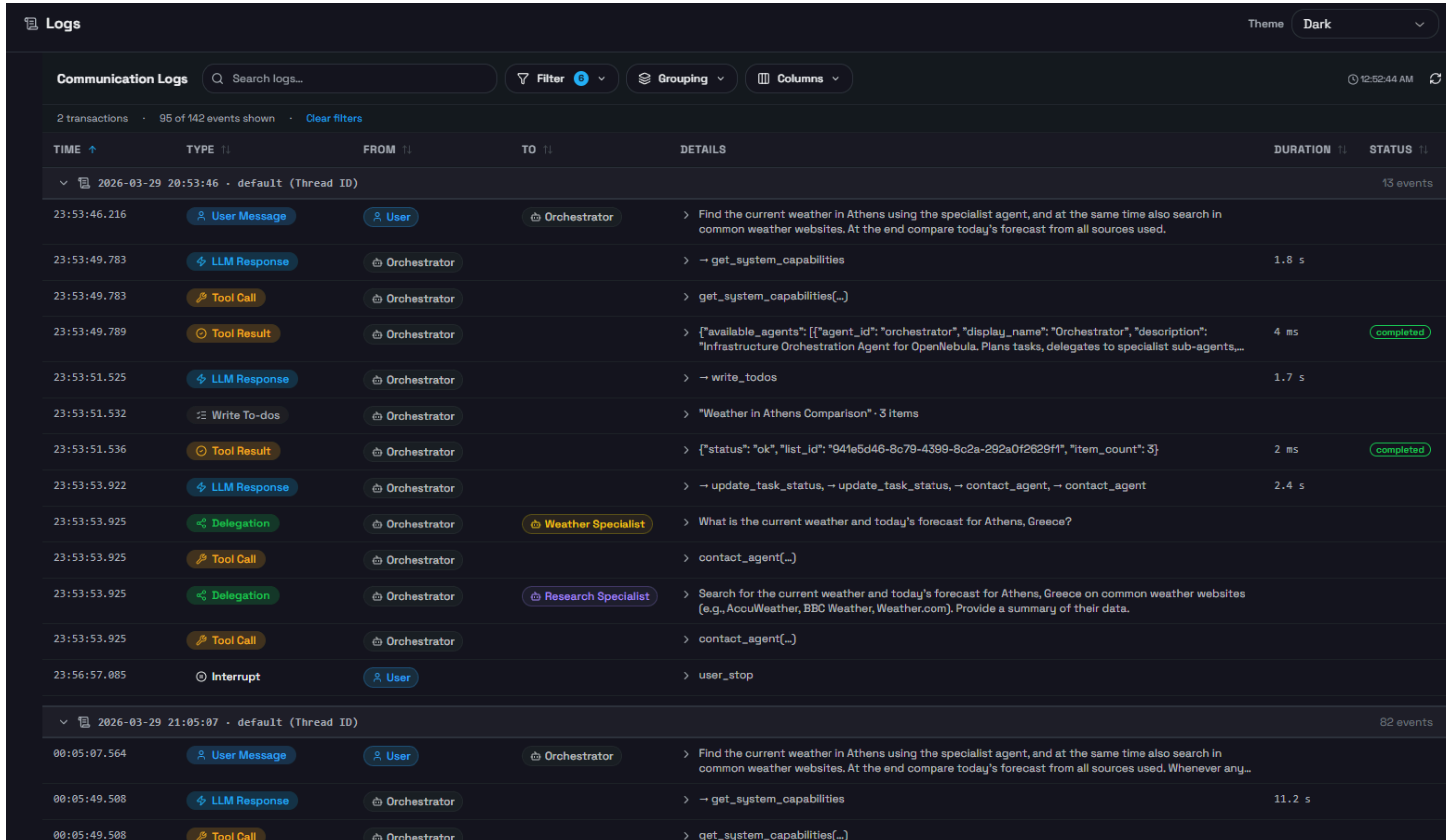
By clicking on the agent names in the Orchestrator's responses, the user is redirected to an agent-to-agent chat page, where they can see the communication between the Orchestrator and the agent whose name they clicked on. They can view the communication between any two agents by changing the Agent 1 and Agent 2 options on the top of the page.



[IMAGE 7.6: The Agent Chats page. The user can view the communication between any 2 agents by selecting them from the drop-down options at the top of the page.]

7.4 Logs page

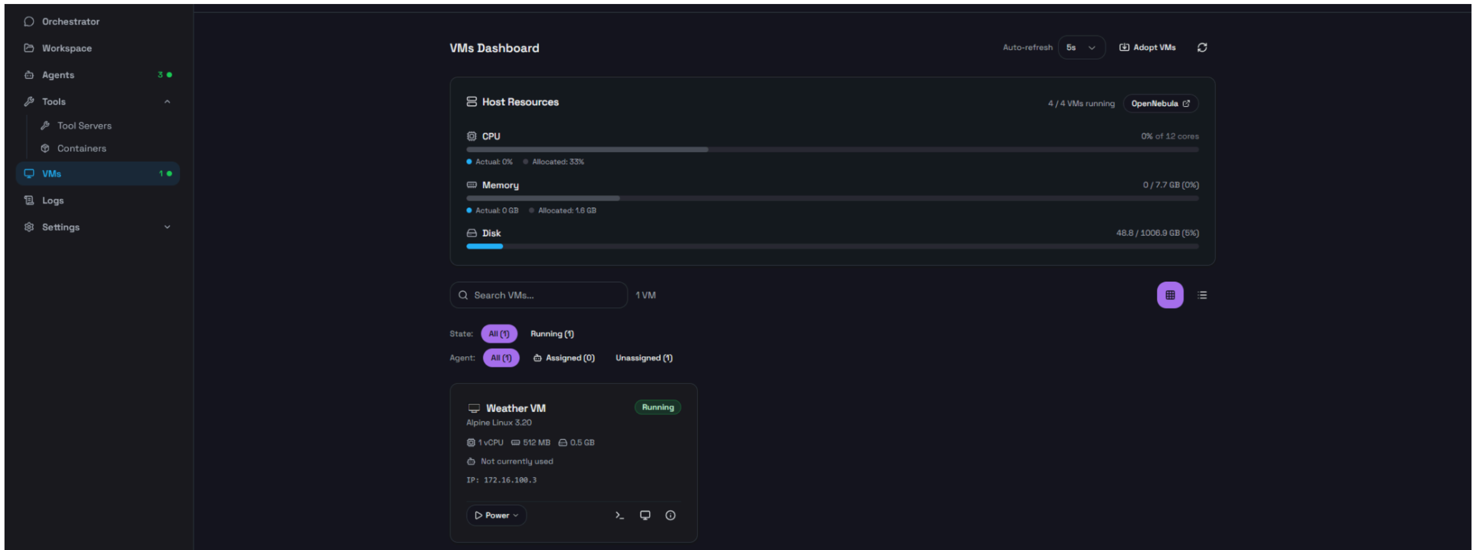
For a more analytic view of the workflow, the user can refer to the project's Logs page, where they can see messages sent to and from agents, tool calls, tool results, task delegations and more.



[IMAGE 7.7: The Project's Logs page.]

7.5 VMs page

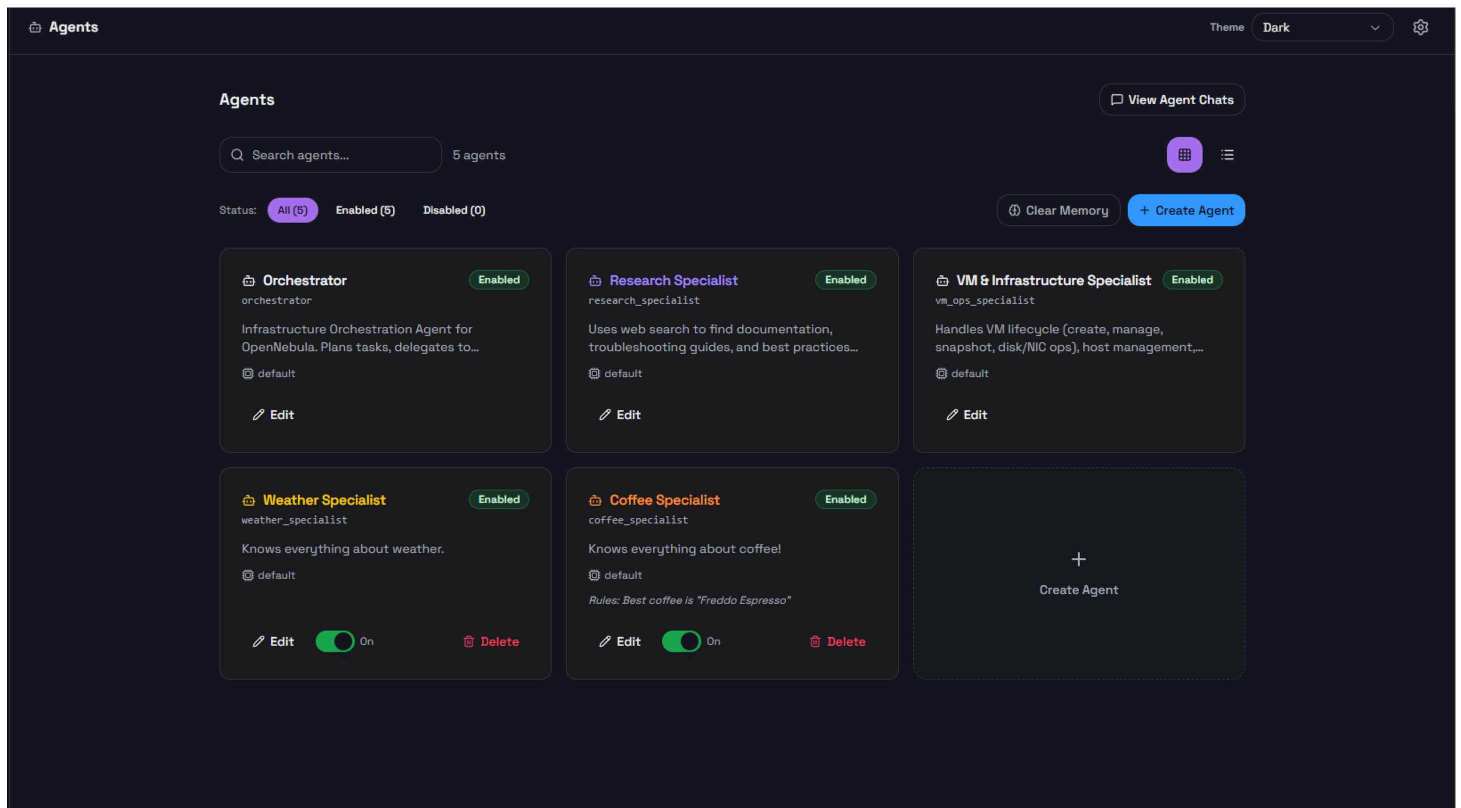
On the VMs page of the project, the user can view the VMs that were created and are being used in the project along with extra info regarding the host resources. They can also manually power on and off a VM of their choice, or connect to it via SSH or VNC.



[IMAGE 7.8: The VMs Dashboard page.]

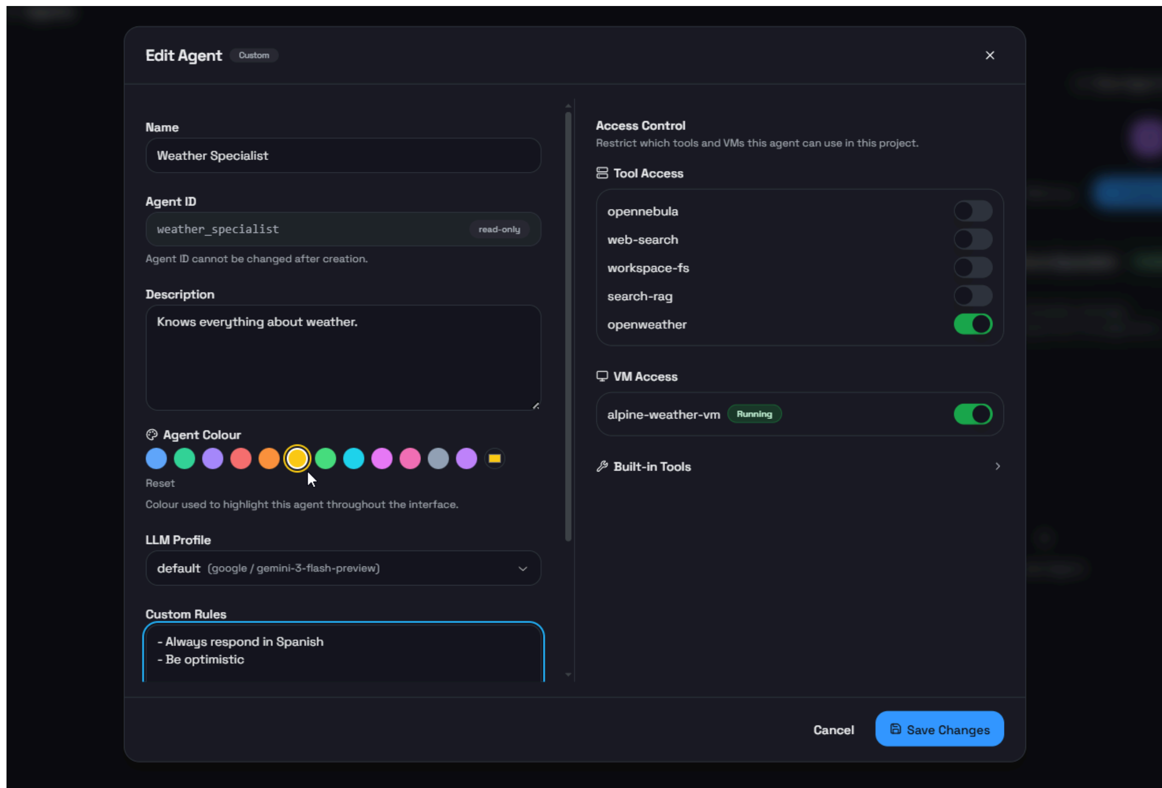
7.6 Agents page

On the project's Agents page, the user can view all the agents that are available in the project and even create their own, custom agents that suit their needs. All custom agents can be created, customized and can be enabled or disabled on-demand by the user. By clicking on the "View Agent Chats" button they can inspect the communication between any pair of agents. The "Clear Memory" button resets all session / memory information for this project.



[IMAGE 7.9: The Agents page.]

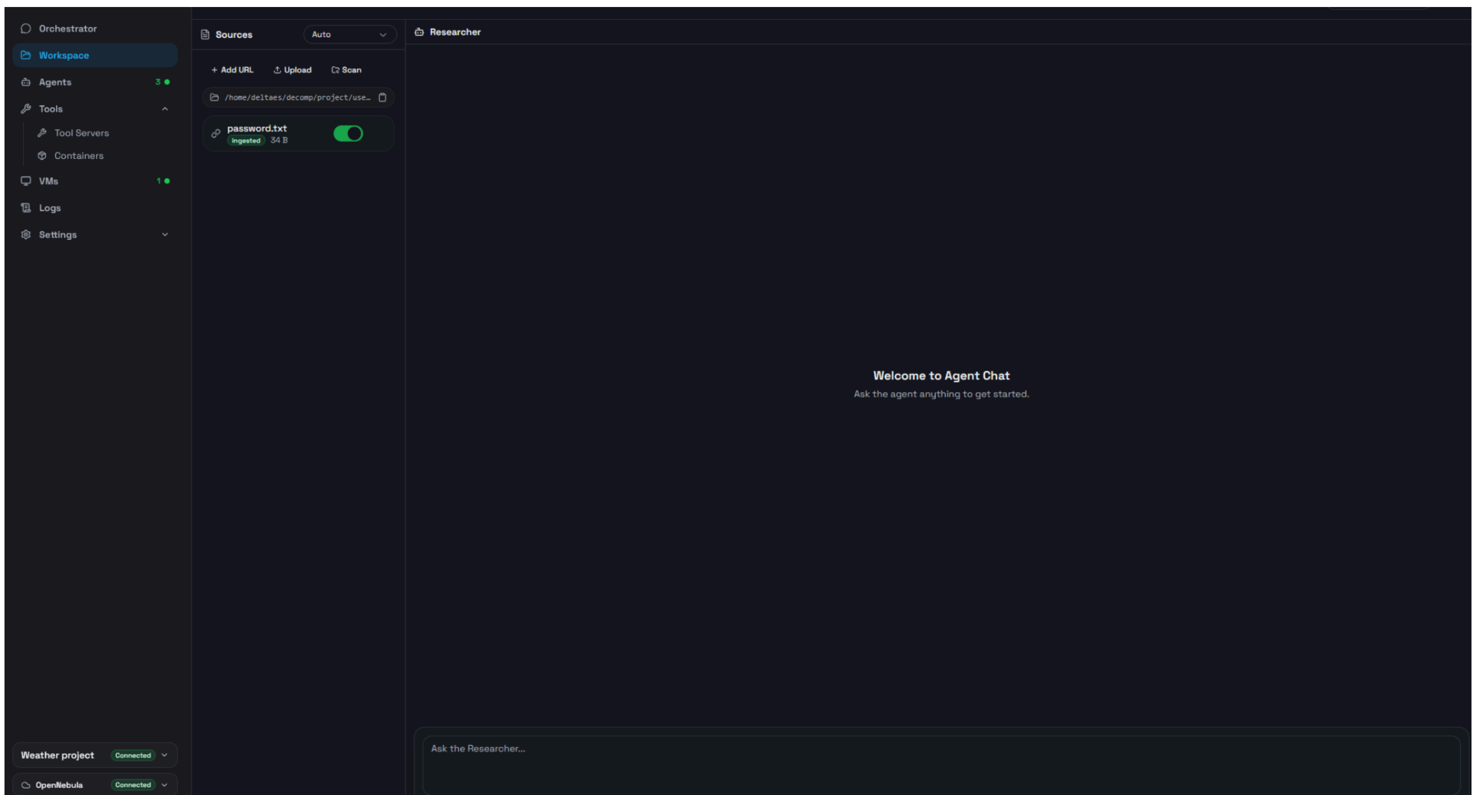
The user can set their own custom rules for their custom agents, configure tools and VM access for better workflow balancing and increased security.



[IMAGE 7.10: The Edit Agent modal.]

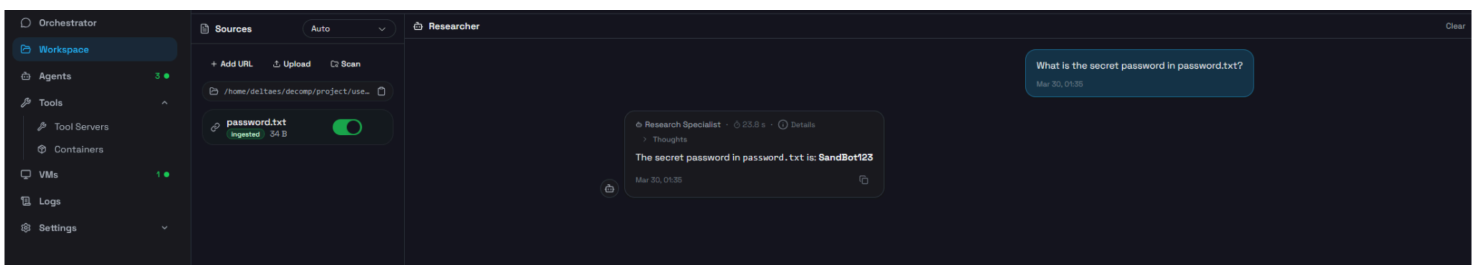
7.7 Workspace page

The Workspace page is the hub for all important files of the project. The user and the agents can transfer files to and from the Workspace (from VMs, downloaded from the internet etc.). All files in the Workspace are automatically added to a RAG, which is by default used as context for the Researcher agent.



[IMAGE 7.11: The Workspace page.]

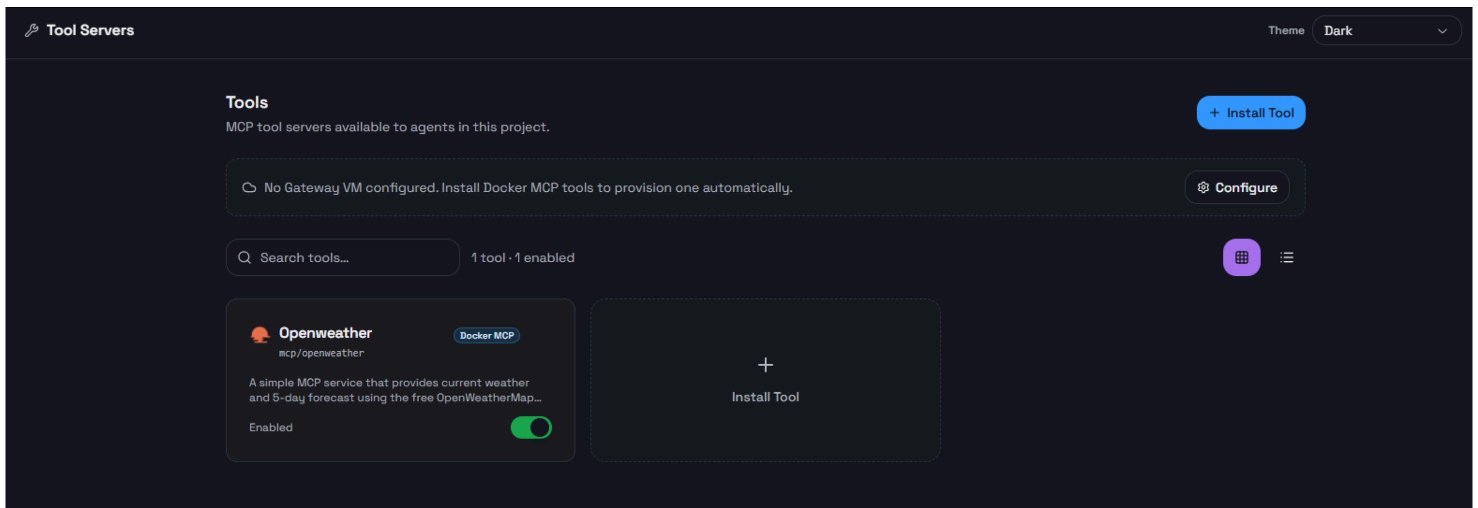
The user can directly chat with the Researcher agent and ask about anything regarding the Workspace's files. The user can also request from the Researcher to find more relevant sources for the project and store them in the Workspace.



[IMAGE 7.12: The Workspace page. Researcher chat.]

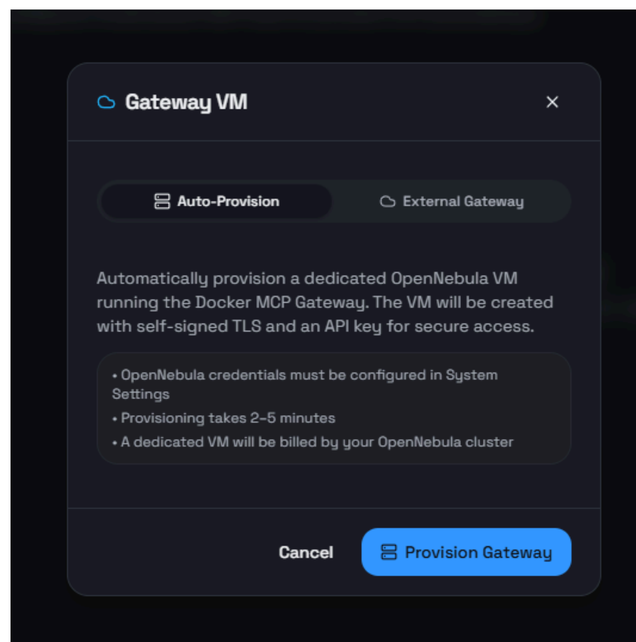
7.8 Tools page

In the Tools page, the user can install and configure custom MCP tools to be used by the project's agents. The tools run under a dedicated VM whose status appears in this page. For each installed tool, the user can choose to enable or disable it, as well as configure it.



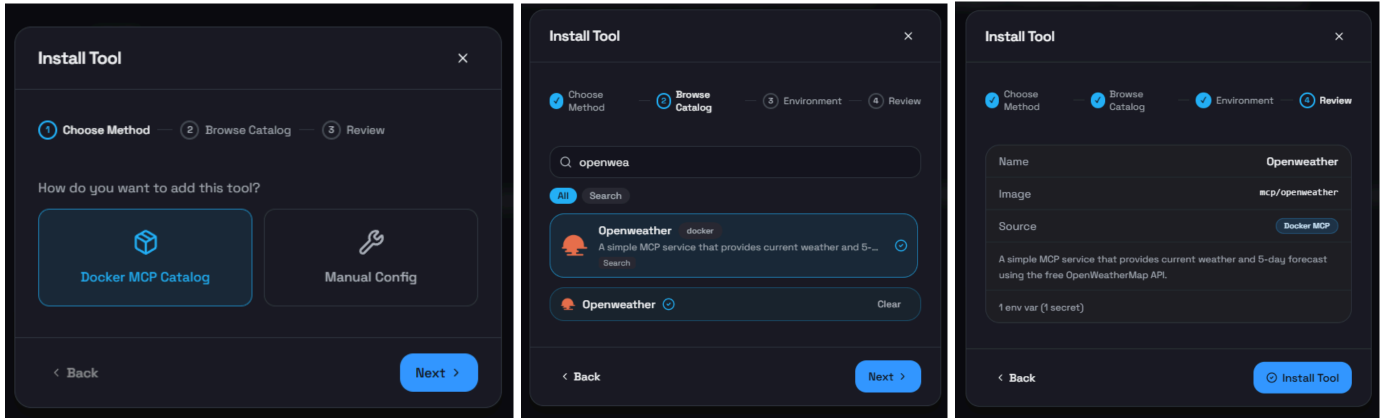
[IMAGE 7.13: The Tools page.]

The Gateway VM, the dedicated VM running the mcp-tools, can be automatically deployed by the system or manually configured by the user.



[IMAGE 7.14: The Gateway VM provision modal.]

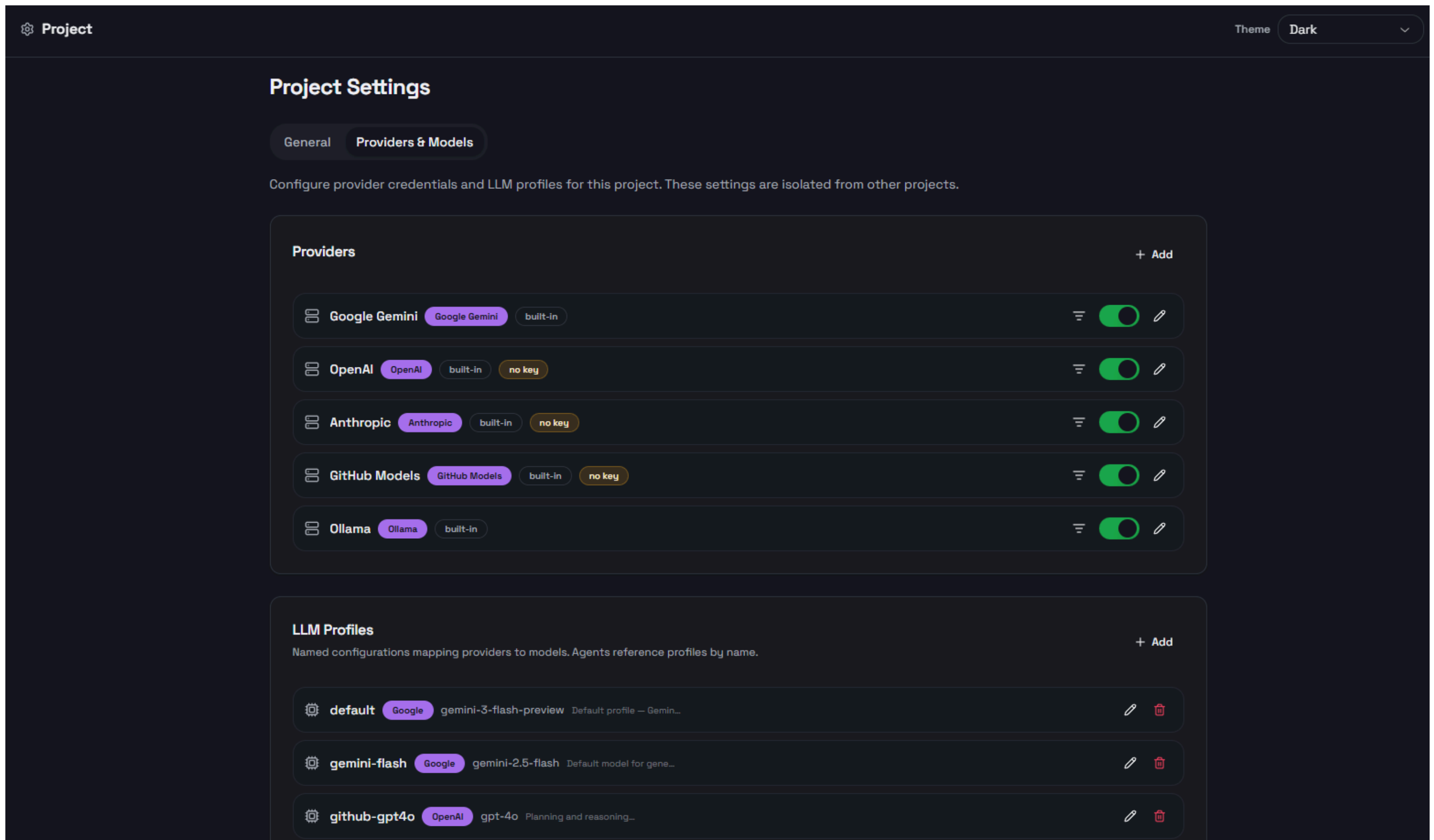
The “Install Tool” modal allows the user to select which docker mcp-tools they want to install. The user can manually configure the needed tools, or browse the Docker MCP Catalog. After selecting and configuring the requested tools the system downloads and installs them to the Gateway VM.



[IMAGE 7.15: Install Tool modal.]

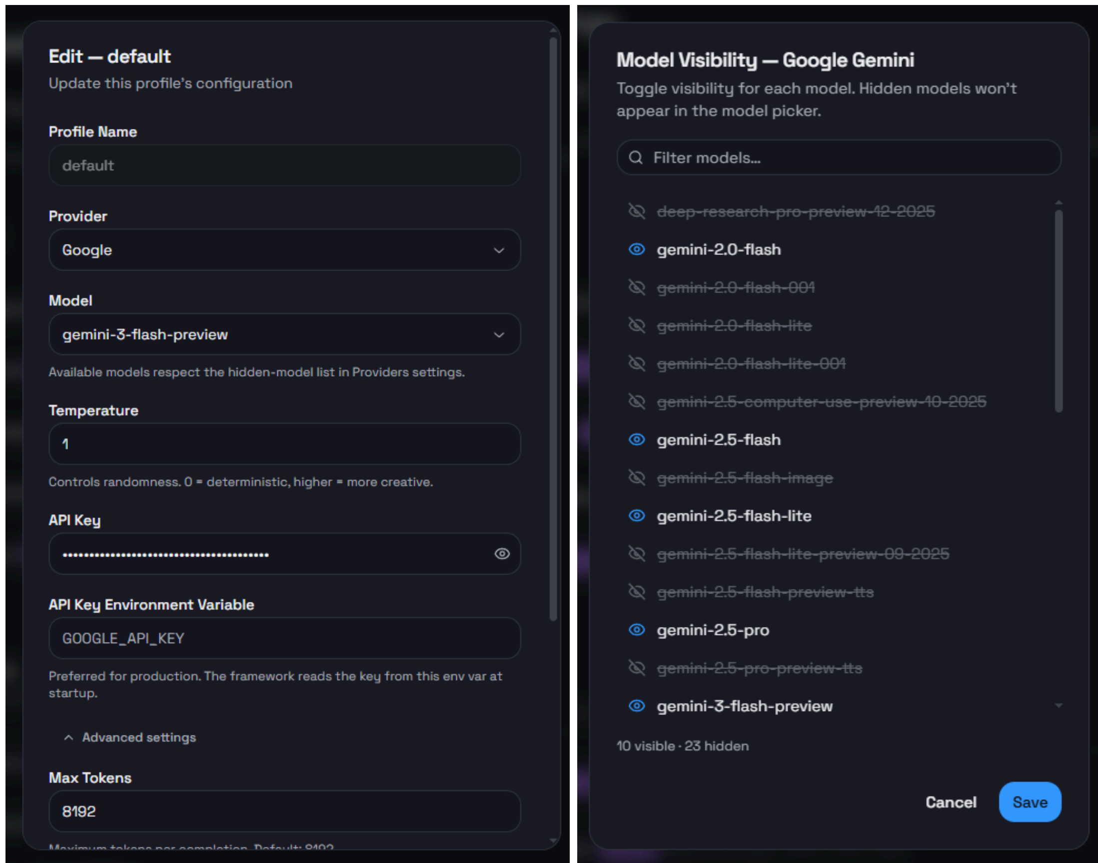
7.9 Settings page

The Project Settings page allows the user to configure settings related to the open project, such as Provider/LLM settings, general agent settings, context budgets etc.

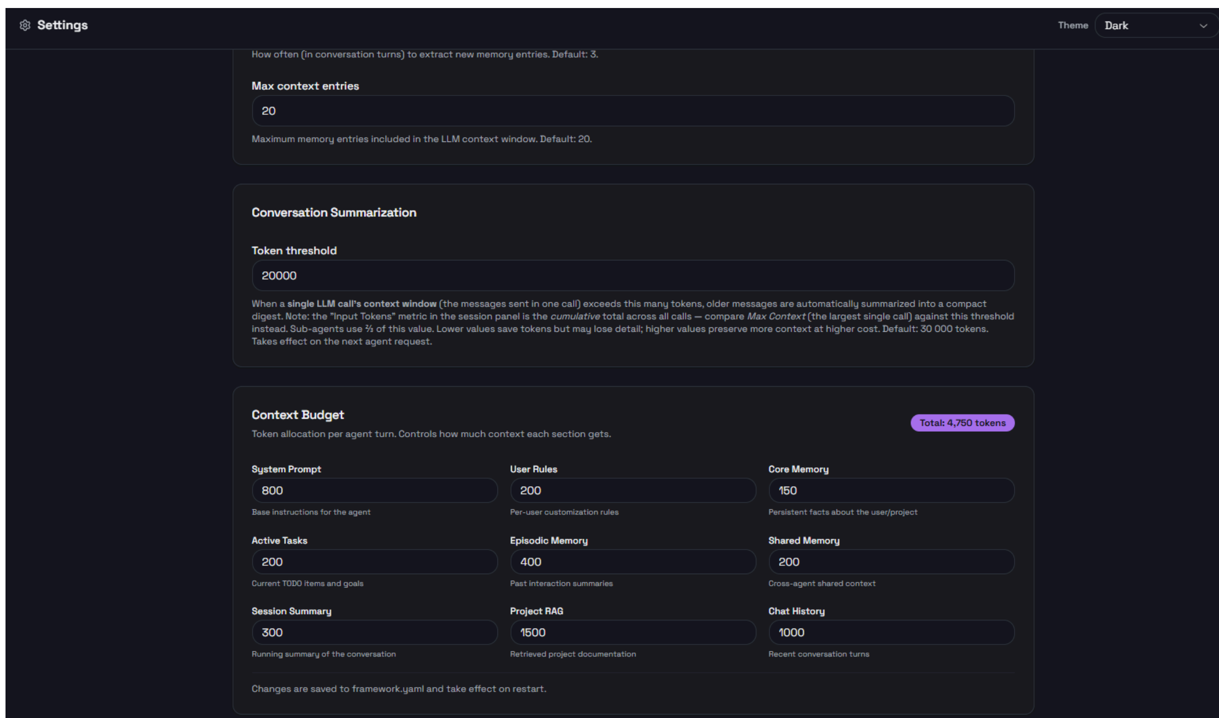


[IMAGE 7.16: The Project Settings page.]

The user can edit the LLM / Provider profiles, configuring attributes related to the API, available models, etc.



[IMAGE 7.17: Project Settings page. LLM Profile and Model Visibility settings.]



[IMAGE 7.18: Project Settings page. Conversation Summarization and Context Budget settings.]

8. CONCLUSION AND FUTURE WORK

As artificial intelligence transitions from isolated, generative conversational models to autonomous, multi-step execution engines, the architectural frameworks that govern these systems must evolve proportionately. This thesis has introduced SandBot (Decomposition & Orchestration Framework), an enterprise-grade Multi-Agent System (MAS) designed to address the critical bottlenecks of context degradation, vendor lock-in, and infrastructure security. This final chapter synthesizes the research findings, provides an analytical review of the current limitations of the implemented architecture, and outlines actionable directions for future research in the field of autonomous agent coordination.

8.1 Thesis Summary and Synthesis of Findings

The primary objective of this research was to demonstrate that the fundamental limitations of monolithic AI agents - specifically the "Memory Wall", the "unreliability tax" of closed-source APIs, and the vulnerability to Arbitrary Code Execution (ACE) - can be systematically mitigated through decoupled, peer-to-peer orchestration and rigorous infrastructure sandboxing. The development of SandBot served as a proof-of-concept for a modular, security-first architecture that prioritizes state integrity and economic sustainability.

The empirical evaluation of the SandBot framework yielded several conclusive findings that directly answer the research questions (RQs) established in Chapter 1:

1. **Decomposition Mitigates Context Degradation (RQ1):** By implementing an Orchestrator-Delegate pattern with Memory Namespace Isolation, SandBot successfully partitioned cognitive load across specialized peers. This architectural choice effectively neutralized the "lost-in-the-middle" phenomenon by ensuring that each specialized agent only processed information relevant to its immediate domain. The evaluation showed that while monolithic agents failed at a 58% rate due to context pollution and instruction drift during complex deployments, SandBot maintained a 94% success rate. This suggests that "cognitive partitioning" is a more effective strategy for long-horizon tasks than simply increasing the raw context window of a single model.

2. **Algorithmic Optimization Neutralizes the Memory Wall (RQ2):** The integration of Tool Description Deduplication achieved an ~47% reduction in Model Context Protocol (MCP) overhead. Combined with the Six-Section Prompt Architecture, this mathematically guaranteed context stability even when integrating high-density infrastructure schemas (such as OpenNebula's). This finding is significant because it proves that the "Memory Wall" is not an insurmountable physical limit, but rather an architectural challenge that can be managed through intelligent data compaction and "just-in-time" tool loading.

3. **Hardware-Level Sandboxing Secures Agency (RQ3):** The programmatic integration of OpenNebula VMs, using custom automated deployment, provided a secure, "fail-closed" environment. The evaluation demonstrated that hardware-isolated execution is not only more secure than container-based alternatives - which share the host kernel - but also robust enough to handle the inherent network volatility of cloud infrastructure through automated recovery protocols. This highlights that for AI to have true agency, it must operate within a boundary that respects traditional security principles of least privilege and complete isolation.

4. **Hybrid Routing and BYOM Optimize TCO (RQ4):** The Bring-Your-Own-Model architecture allowed for an economic optimization strategy. By routing iterative sub-tasks (like code linting or formatting) to locally hosted open-weight models (e.g., Llama-3) while reserving proprietary models for high-level strategy, the framework proved that autonomous project execution can be both sovereign and cost-effective. This effectively dodges the "unreliability tax" of managed APIs, where every hallucination or retry incurs a direct financial cost, making agentic workflows viable for large-scale enterprise adoption.

8.2 Limitations and Critical Analysis

Despite the successful implementation and validation of the SandBot framework, several inherent constraints were identified during the research that define the current ceiling of the system's performance:

- **The Orchestration Latency Tax:** While decomposing tasks into specialized agents improves accuracy and context management, it introduces a significant "latency tax." Every handoff between the Orchestrator and a Specialist requires a full LLM generation cycle, middleware interceptors (e.g., pre_llm_call), and network transmission. In scenarios requiring near-instantaneous feedback - such as real-time system monitoring or high-frequency trading - this cumulative asynchronous delay can be prohibitive. The serialized nature of the ReAct loops across multiple agents creates a bottleneck that monolithic scripts do not share.

- **Cognitive Barrier to Configuration:** The "Atomic Agent" model, while providing exceptional immutability and security through YAML-based configuration, presents a steep learning curve. For a non-technical project manager, defining precise token budget keys, MCP transport protocols, and memory namespaces requires a level of architectural expertise that current commercial platforms abstract away. This creates a "usability-security trade-off" where the framework's robustness is inversely proportional to its ease of setup, potentially limiting its adoption to organizations with high engineering maturity.

- **Graph Scaling and Retrieval Noise:** As project timelines extend into thousands of interaction turns, the Graphiti knowledge graph grows in complexity. While the temporal-decay scoring formula successfully prunes trivial facts, the risk of "retrieval noise" increases. Over time, an agent may retrieve conflicting architectural decisions or "stale" state information from early in the project that is no longer valid but remains prominent in the graph. Furthermore, the computational cost of performing complex graph traversals and memory extractions on every terminal response adds an overhead that scales with the project's duration.

- **State Reconciliation Complexity:** In a distributed environment, ensuring that the AI's "mental model" of the infrastructure perfectly matches the physical state of the VMs is difficult. If an external event (e.g., a hypervisor crash) alters the environment without the agent's knowledge, the framework's "fail-closed" mechanisms must be perfectly synchronized to prevent the agent from proceeding with a hallucinated state. The current Three-Layered Provisioning addresses this, but it adds significant complexity to the codebase and requires strict adherence to asynchronous waiting periods.

8.3 Future Work and Research Directions

The findings and limitations identified in this thesis establish a foundation for several promising avenues of future research in the field of autonomous AI coordination:

- **Reinforcement Learning for Dynamic Routing:** Future iterations of SandBot should move away from static agent registries toward a Reinforcement Learning (RL) based "Meta-Orchestrator." By treating model and tool routing as a multi-armed bandit problem, the framework could autonomously learn which model-specialist combinations yield the highest success rates for specific task types. This would allow the system to self-optimize for both cost and latency, dynamically shifting workloads to cheaper models as they demonstrate increased competency in a specific domain.

- **Federated MCP and Multi-Cloud Orchestration:** Currently, SandBot is optimized for OpenNebula. Expanding the framework to support a federated Model Context Protocol would allow for multi-cloud orchestration. This research would focus on how security tokens and "contextual identities" can be passed securely between different cloud providers (e.g., AWS, Azure, GCP) while maintaining the hardware-level isolation and lifecycle guarantees established in this thesis.

- **Self-Healing Sandbox Mechanisms:** Building upon the MicroVM security layer, future research could integrate "self-healing" supervisor agents within the execution playground. These supervisors would use eBPF or similar low-level monitoring tools to detect build failures, kernel panics, or configuration drift in real-time. Instead of returning an error to the main agent, the supervisor could autonomously apply micro-snapshots or patches, reducing the cognitive burden on the primary Orchestrator and further decreasing the "unreliability tax" of infrastructural debugging.

- **Natural Language Agent-as-Code (AIaC):** To address the configuration barrier, research into "Agent-as-Code" generation is vital. This would involve a high-level interface where a user describes a specialist’s requirements in natural language, and a specialized "Architect Agent" autonomously generates the 4-component Atomic Agent definition (YAMLs and Prompts). This would bridge the gap between technical rigor and user accessibility, allowing domain experts to bootstrap specialized MAS networks without deep knowledge of the underlying framework.

- **Asynchronous Parallel Reasoning (APR):** To mitigate the "latency tax", research into parallelizing the reasoning paths of agents - where multiple sub-agents explore different solution branches simultaneously before converging back to the Orchestrator - could improve execution speed. This would require advanced state-merging algorithms to handle conflicting observations from different solution branches.

In conclusion, SandBot demonstrates that by treating AI agents as modular, secure, and cooperative specialists rather than monolithic conversational tools, we can build a new generation of autonomous systems. These

systems are capable of executing more complex technical projects with a level of precision, security, and economic sustainability that single-agent architectures simply cannot achieve. This research provides the blueprint for a future where autonomous AI is not just a creative collaborator, but a reliable and secure infrastructural force.

REFERENCES

[1] Narek Maloyan and Dmitry Namiot, "Prompt Injection Attacks on Agentic Coding Assistants: A Systematic Analysis of Vulnerabilities in Skills, Tools, and Protocol Ecosystems", arXiv preprint arXiv:2601.17548, Jan. 2026.

[2] OpenNebula Systems, "White Paper - OpenNebula AI Factory Reference Architecture", OpenNebula.io, 2026. [Online]. Available: https://opennebula.io/

[3] Shamsher Khan, "Decomposing Docker Container Startup Performance: A Three-Tier Measurement Study on Heterogeneous Infrastructure", arXiv preprint arXiv:2602.15214, Feb. 2026.

[4] Wiz Research, "Leaky Vessels: Deep Dive on Container Escape Vulnerabilities (CVE-2024-21626)", Wiz Blog, Feb. 2024. [Online]. Available: https://www.wiz.io/blog/leaky-vessels-container-escape-vulnerabilities

[5] Northflank, "Ephemeral sandbox environments [2026 guide]", Northflank Blog, 2026. [Online]. Available: https://northflank.com/blog/ephemeral-sandbox-environments

[6] Sotiropoulos, D. N. (2024). Development of the "AIRS-x" variant. In The Convergence of Autonomous Cyber Defense and Biologically Inspired Artificial Intelligence: A Longitudinal Study of Software Vulnerability and Adaptive Orchestration.

[7] Avgerinos, T., Cha, S. K., Lim Tze Hao, B., & Brumley, D. (2011). AEG: Automatic Exploit Generation. Proceedings of the Network and Distributed System Security Symposium (NDSS 2011).

[8] Hana Derouiche, Zaki Brahmi, Haithem Mazeni, "Agentic AI Frameworks: Architectures, Protocols, and Design Challenges", arXiv preprint arXiv:2508.10146, Aug. 2025.

[9] Anthropic, "Specification - Model Context Protocol", Model Context Protocol, Nov. 2025. [Online]. Available: https://modelcontextprotocol.io/specification/2025-11-25

[10] Vallikranth Ayyagari, "Model Context Protocol for Agentic AI: Enabling Contextual Interoperability Across Systems", ResearchGate, 2025.

[11] Avgerinos, T., Brumley, D., Davis, J., et al. (2018). The Mayhem Cyber Reasoning System. IEEE Security & Privacy Magazine.

[12] N. N., "Towards a Theory of Interoperability of Software Systems", ResearchGate, 2024.

[13] N. F. Liu et al., "Lost in the Middle: How Language Models Use Long Contexts", Transactions of the Association for Computational Linguistics, 2023.

[14] Zhenyu Zhang et al., "Found in the Middle: How Language Models Use Long Contexts Better via Plug-and-Play Positional Encoding", NeurIPS, 2024.

[15] Sotiropoulos, D. N. (2024). The LYRICEL Framework: Rule-augmented artificial intelligence-empowered cultural E-learning with GPT and machine learning. (Part of the 3WC-GBNRS++ research initiative).

[16] Minsoo Kim et al., "EpiCache: Episodic KV Cache Management for Long Conversational Question Answering", arXiv preprint arXiv:2509.17396, Sep. 2025.

[17] Zilliz, "GraphRAG Explained: Enhancing RAG with Knowledge Graphs", Medium, 2025. [Online]. Available: https://medium.com/@zilliz_learn/graphrag-explained-enhancing-rag-with-knowledge-graphs-3312065f99e1

[18] Guanzhong Pan et al., "A Cost-Benefit Analysis of On-Premise Large Language Model Deployment: Breaking Even with Commercial LLM Services", arXiv preprint arXiv:2509.18101, Sep. 2025.

[19] Gartner Research, "AI Lock-In: Why Skill Loss Puts Your Workforce at Risk", Gartner, 2024. [Online]. Available: https://www.gartner.com/en/articles/ai-lock-in

[20] Accenture, "Sovereign AI: Own your AI future", Accenture Report, 2025.

[21] Abhishek and Dr. Vikas Siwach, "Evaluating Vendor Lock-In and Service Availability Risks in Multi-Cloud Deployments", International Journal of Innovative Research in Technology (IJIRT), 2018.

[22] Shunyu Yao et al., "ReAct: Synergizing Reasoning and Acting in Language Models", arXiv preprint arXiv:2210.03629, 2022.

[23] Sirui Hong et al., "MetaGPT: Meta Programming for a Multi-Agent Collaborative Framework", arXiv preprint arXiv:2308.00352, 2023.

[24] Yubin Kim et al., "Towards a Science of Scaling Agent Systems", arXiv preprint arXiv:2512.08296, Dec. 2025.

[25] Jen-Tse Huang et al., "On the Resilience of LLM-Based Multi-Agent Collaboration with Faulty Agents", arXiv preprint arXiv:2408.00989, 2024.

[26] Chen Han et al., "Conformity Dynamics in LLM Multi-Agent Systems: The Roles of Topology and Self-Social Weighting", ResearchGate, 2025.

[27] Firecrawl, "AI Agent Sandbox: How to Safely Run Autonomous Agents in 2026", Firecrawl Blog, 2026. [Online]. Available: https://www.firecrawl.dev/blog/ai-agent-sandbox

[28] LangChain, "LangGraph: Agent Orchestration Framework for Reliable AI Agents", LangChain Documentation, 2025. [Online]. Available: https://www.langchain.com/langgraph

[29] Checkmarx, "11 Emerging AI Security Risks with MCP (Model Context Protocol)", Checkmarx Blog, Nov. 2025. [Online]. Available: https://checkmarx.com/zero-post/11-emerging-ai-security-risks-with-mcp-model-context-protocol/

[30] Jitang Li et al., "Memory, Consciousness and Large Language Model", arXiv preprint arXiv:2401.02509, 2024.

APPENDICES

Appendix A: Atomic Agent Configuration

This appendix provides the annotated YAML schemas for the vm_ops_specialist, illustrating the 4-component Atomic Agent design.

A.1 Role YAML (role.yaml)

```yaml
id: vm_ops_specialist
name: Infrastructure Specialist
immutable: true
provider:
  profile: local-llama3-70b
  base_url: "http://ollama:11434"
  temperature: 0.1
budget:
  system_prompt: 1000
  memory: 1500
  rag: 1500
  tools: 4000
  conversation: 2000
  scratchpad: 1000
```

A.2 Tool YAML (tools.yaml)

```yaml
mcp_servers:
  - name: one-mcp
    transport: stdio
    command: "python"
    args: ["-m", "one_mcp_server"]
    env:
      ONE_XMLRPC: "http://opennebula:2633/RPC2"
    filter_rules:
      include: ["vm_*", "oneflow_*"]
      exclude: ["host_*", "zone_*"]
```

Appendix B: Tool Description Deduplication (MCP)

Below is a comparison of a raw Model Context Protocol (MCP) tool schema versus the deduplicated internal pointer format used by the SandBot ContextAssembler.

B.1 Raw JSON Schema (Abridged)

```
{
  "name": "vm_instantiate",
  "description": "Create a new VM instance",
  "parameters": {
    "type": "object",
    "properties": {
      "template_id": { "type": "integer" },
      "vm_name": { "type": "string" },
      "network_config": {
        "type": "object",
        "properties": {
          "ip": { "type": "string" },
          "vnet_id": { "type": "integer" }
        }
      }
    }
  }
}
```

B.2 Deduplicated "Pointer" Format

```
{
  "name": "vm_instantiate",
  "parameters": {
    "$ref": "#/$defs/vm_base_params",
    "properties": {
      "network_config": { "$ref": "#/$defs/network_config_schema" }
    }
  }
}
```

Appendix C: Sample Execution Traces

C.1 Orchestrator State Transition (Arize Phoenix Extract)

```
[TIMESTAMP: 2026-03-29T10:15:22.001Z]
[NODE: orchestrator_planner]
[INPUT: "Deploy the load balancer for the project."]
[THOUGHTS: "Project requires a load balancer. I need to delegate this to the vm_ops_specialist."]
[ACTION: CALL task(agent="vm_ops_specialist", objective="Provision an Nginx load balancer VM.")]
[TASK_ID: 550e8400-e29b-41d4-a716-446655440000]
```

C.2 Specialist Execution Log (Middleware Interception)

```
[TIMESTAMP: 2026-03-29T10:15:45.312Z]
[AGENT: vm_ops_specialist]
[PRE_TOOL_HOOK: Intercepting tool 'vm_instantiate']
[SECURITY_CHECK: Human-in-the-loop (HITL) manual approval triggered.]
[STATUS: Waiting for user signature…]
[USER_SIGNATURE: GRANTED (admin_01)]
[EXECUTING: onevm instantiate 104 --name lb-node-01]
[POST_TOOL_HOOK: Compressing 12kb XML response to 140b JSON summary.]
```

Appendix D: The Six-Section Prompt Template

This appendix provides the master template used by the ContextAssembler to construct the context window. It illustrates how the budget keys from the Role YAML are applied.

D.1 Context Construction Template (Markdown)

```
# SECTION 1: SYSTEM IDENTITY
{{system_prompt}}

# SECTION 2: EPISODIC MEMORY
{{memory_context}}

# SECTION 3: GROUNDING (RAG)
{{rag_context}}

# SECTION 4: CONVERSATION HISTORY
{{conversation_history}}

# SECTION 5: AVAILABLE CAPABILITIES (TOOLS)
{{deduplicated_mcp_tools}}

# SECTION 6: WORKING SCRATCHPAD
{{active_reasoning_steps}}
```

Appendix E: Inter-Agent Communication Protocol

This appendix details the JSON "envelope" used for peer-to-peer delegation and cross-conversation notifications, managed by the SessionManager.

E.1 Task Delegation Envelope


```
{
  "header": {
    "source_agent": "orchestrator",
    "target_agent": "vm_ops_specialist",
    "request_id": "req_88219",
    "timestamp": "2026-03-29T11:00:01Z"
  },
  "payload": {
    "objective": "Verify if the database VM is reachable on port 5432.",
    "namespace": "shared_project_v1",
    "constraints": {
      "timeout": 60,
      "retries": 3
    }
  }
}
```

Appendix F: OpenNebula Lifecycle State Mapping

A technical mapping showing how internal "SandBot" states correlate with physical OpenNebula VM states (onevm show outputs).

SandBot State	OpenNebula State	Lifecycle Phase
PROVISIONING	PENDING / BOOT	Hypervisor Allocation
INITIALIZING	RUNNING (wait for ssh)	Cloud-Init / Setup
ACTIVE	RUNNING	Tool-calling ready
TERMINATING	DONE	Clean-up / Cleanup Registry
ERROR	FAILURE / UNKNOWN	Automatic Recovery Trigger

Appendix G: Graphiti Memory Extraction Sample

An example of the transformation logic performed by the MemoryExtractionMiddleware, converting raw agent dialogue into structured Graph nodes.

Input Phrase: "I have successfully configured the load balancer at 192.168.1.50 using Nginx."

Extracted Triplets:

1. (LoadBalancer, has_ip, "192.168.1.50")
2. (LoadBalancer, utilizes, Nginx)
3. (vm_ops_specialist, performed, ConfigurationTask)