MAMOS Research Round 2 Report

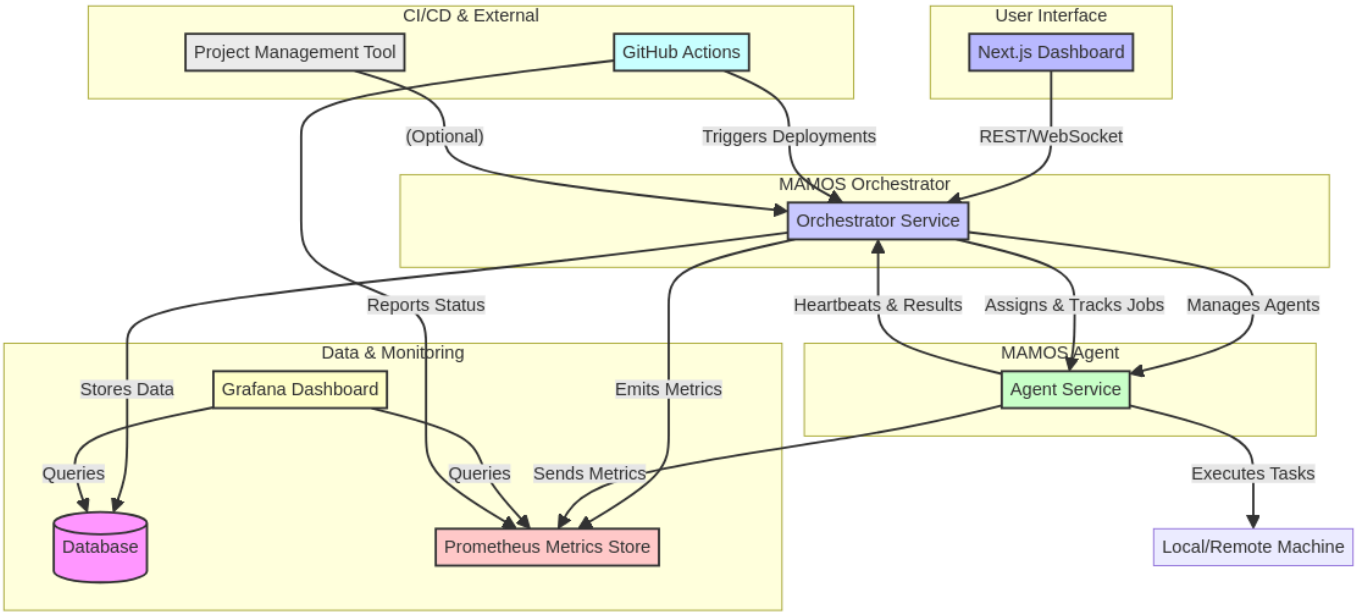
Author: Manus AI **Date:** Oct 14, 2025 PDT

1. Introduction

This report provides a comprehensive technical research and design plan for the **MAMOS (Manus Agent Management & Orchestration System)** platform. Following the initial research on monitoring, this document expands the scope to cover the full, three-layer system as requested, encompassing the **Orchestrator Layer**, the **Agent Layer**, and the **Monitoring & Visualization Layer**. The goal is to present a complete, implementable architecture and a step-by-step MVP plan that aligns with the vision of MAMOS as an intelligent orchestration and monitoring system for distributed agents.

2. Updated System Architecture Diagram

The updated system architecture for MAMOS is designed to be modular, scalable, and container-based. It clearly defines the roles and interactions of the three core layers:



Architecture Components:

- **User Interface (Next.js Dashboard):** The web-based interface for users to manage agents, create and monitor jobs, and view system metrics.
- **MAMOS Orchestrator (NestJS):** The central brain of the system, responsible for agent management, job orchestration, and data persistence.
- **MAMOS Agent (Python):** Lightweight clients that run on local or remote machines, execute tasks, and report back to the orchestrator.

- **Data & Monitoring:** A suite of tools for collecting, storing, and visualizing metrics, including Prometheus for metrics storage and Grafana for dashboarding.
- **CI/CD & External:** Integration with GitHub Actions for automated testing and deployment, and optional integration with project management tools.

3. Recommended Tech Stack

The technology stack for MAMOS is chosen to be modern, scalable, and maintainable, leveraging open-source tools and containerization for reproducibility.

Layer	Technology	Justification
Orchestrator Layer	**NestJS (Node.js)**	A progressive Node.js framework for bu reliable, and scalable server-side appli architecture is ideal for MAMOS.
	Prisma	A modern database toolkit for TypeScri simplifies database access and manage
	SQLite / PostgreSQL	SQLite for MVP due to its simplicity and PostgreSQL for production for its robus
	Redis	An in-memory data store, used as a me caching, providing high performance fo memory for MVP).
Agent Layer	**Python 3.11+**	A versatile and widely-used language w support for system-level tasks, scriptin ecosystem of libraries.
	Typer / Click	For creating a clean and maintainable c interface for the agent.
	Requests / websockets	For communication with the Orchestra
Monitoring & Visualization	**Next.js**	A React framework for production-grad providing a rich user experience for the
	Prometheus	A leading open-source monitoring and designed for reliability and scalability.
	Grafana	The open-source platform for monitori observability, allowing for the creation dashboards.
Infrastructure	**Docker / Docker Compose**	For containerizing all components of th ensuring consistency across developm production environments.

4. MVP Implementation Plan (Step-by-Step)

This section outlines a phased approach to implementing the MAMOS MVP within the `aladdin-sandbox-MAMOS` repository, focusing on incremental value delivery.

Phase 1: Orchestrator Setup

1. **Initialize NestJS Project**: Create the basic NestJS application structure for the Orchestrator.
2. **Database Integration (Prisma)**: Set up Prisma with SQLite (for MVP) and define initial schemas for Agents, Jobs, and Users.
3. **Authentication Module**: Implement JWT-based authentication for both Dashboard users and Agents, including user registration and login endpoints.
4. **Agent Management Endpoints**: Develop REST APIs for Agent registration (using one-time tokens), heartbeat reception, and status updates.
5. **Job Management Endpoints**: Create REST APIs for job creation, assignment, status updates, and result storage.
6. **WebSocket Gateway**: Implement a WebSocket gateway for real-time communication with the Dashboard and Agents (e.g., live status updates, job progress).
7. **Basic Logging & Error Handling**: Set up centralized logging and robust error handling.

Phase 2: Agent Prototype

1. **Initialize Python Project**: Create the basic Python application structure for the Agent.
2. **Configuration Management**: Implement `agent.yaml` parsing for server URL, token, and other settings.
3. **Orchestrator Connection**: Develop logic to connect to the Orchestrator via WebSocket or REST, using the registration token.
4. **Heartbeat Mechanism**: Implement periodic heartbeat sending (status, CPU/RAM/OS info) to the Orchestrator.
5. **Task Execution**: Develop a secure command execution module with a strict whitelist for allowed commands.
6. **Result Reporting**: Implement logic to send task results and logs back to the Orchestrator.
7. **Background Service Integration**: Provide scripts/instructions for running the Agent as a background service (systemd for Linux, Task Scheduler for Windows).

Phase 3: Metrics & Monitoring Integration

1. **Prometheus & Grafana Setup**: Deploy Prometheus and Grafana using Docker Compose within the `infra/docker` directory.
2. **Orchestrator Instrumentation**: Integrate Prometheus client libraries into the Orchestrator to expose metrics (e.g., active agents, job queue size, API request rates).

3. **Agent Instrumentation**: Integrate Prometheus client libraries into the Agent to expose system metrics (CPU, RAM, disk usage) and agent-specific metrics.
4. **GitHub Actions Metrics**: Configure GitHub Actions workflows to report build/test status and duration, either directly to Prometheus or via the Orchestrator.
5. **Grafana Dashboard Creation**: Design and implement initial Grafana dashboards to visualize key metrics: active agents, system resource usage, job queue status, CI/CD build results, and task success rates.

Phase 4: Unified Dashboard & Release

1. **Next.js Dashboard Development**: Build the Next.js application, including:
   - **Authentication**: Integrate with the Orchestrator's JWT authentication.
   - **Agent View**: Display registered agents, their status, and system metrics (from Orchestrator/Grafana).
   - **Job View**: List jobs, their status, and results. Allow for job creation.
   - **Monitoring View**: Embed Grafana dashboards or recreate key visualizations natively using charting libraries (e.g., Recharts) and data from the Orchestrator.
   - **Real-time Updates**: Utilize WebSockets from the Orchestrator for live data updates.
2. **CI/CD Workflow Refinement**: Ensure `test-build.yml`, `release.yml`, and `docker-publish.yml` are fully functional for all components (Orchestrator, Dashboard, Agent).
3. **Documentation Finalization**: Complete `README.md`, `ARCHITECTURE.md`, `DEPLOYMENT.md`, `AGENT.md`, `SECURITY.md`, `CONTRIBUTING.md`, `CODE_OF_CONDUCT.md`.
4. **One-Click Installers**: Finalize `install_linux.sh`, `install_win.ps1`, `install_agent_linux.sh`, `install_agent_win.ps1`.
5. **GitHub Release**: Prepare and publish a GitHub Release with a ZIP artifact containing all source code, documentation, and installation scripts.

5. CI/CD Integration Proposal

Effective CI/CD integration is crucial for automating the development lifecycle and feeding relevant metrics into the MAMOS monitoring system. The proposal focuses on leveraging GitHub Actions to provide build, test, and deployment data.

5.1. Reporting Build/Test Data to Prometheus or Orchestrator

There are two primary approaches for GitHub Actions to report data:

1. **Directly to Prometheus (Recommended for Metrics)**:

- **Mechanism**: After a build or test job completes in GitHub Actions, a step can be added to push relevant metrics (e.g., build duration, test success/failure count, code coverage percentage) to a Prometheus Pushgateway. Prometheus can then scrape this Pushgateway. Alternatively, if the Orchestrator exposes a Prometheus-compatible endpoint, GitHub Actions can make a direct HTTP request to that endpoint with the metrics.
- **Implementation Hint**: Use a dedicated GitHub Action or a simple `curl` command within the workflow to send data to the Pushgateway or Orchestrator's metrics endpoint.
- **Example (Pushgateway)**:

2. **Directly to Orchestrator (Recommended for Status/Logs)**:
   - **Mechanism**: GitHub Actions can make a REST API call to the Orchestrator to report the status of a build, test, or deployment, along with any relevant logs or artifacts. The Orchestrator would then store this information in its database and potentially emit its own metrics to Prometheus.
   - **Implementation Hint**: Create a dedicated API endpoint in the Orchestrator (e.g., `/api/ci/status`) that GitHub Actions can call with a payload containing the workflow run details, status, and links to logs.

5.2. Displaying Deployment Metrics in the Dashboard

Deployment metrics will be displayed in the MAMOS Next.js Dashboard, leveraging data from both Prometheus/Grafana and the Orchestrator's database.

- **Grafana Embedding**: For a quick MVP, relevant Grafana panels or entire dashboards showing deployment frequency, success rates, and duration can be embedded directly into the Next.js Dashboard using iframes. This provides rich visualization without duplicating effort.
- **Native Dashboard Integration**: For a more seamless experience, the Next.js Dashboard can directly query the Orchestrator's API (which in turn might query Prometheus or its own database) to fetch deployment-related data. This data can then be visualized using charting libraries within the Next.js application.
  - **Key Metrics to Display**: Deployment frequency, success/failure rates per project, average deployment time, rollback count, and environment status (e.g., `dev`, `staging`, `production`).
- **CI/CD Workflow Status**: A dedicated section in the Dashboard can show the real-time status of GitHub Actions workflows for each project within the monorepo, indicating if builds are passing, failing, or in progress.

6. Security Considerations

Security is paramount for a system like MAMOS, especially given its control over distributed agents and execution of commands. Key security aspects must be addressed from the outset.

6.1. Agent Authentication Model

- **One-Time Registration Tokens**: Agents should register with the Orchestrator using a short-lived, one-time token. This token is generated by the Orchestrator and provided to the agent out-of-band (e.g., manually copied by the user). Upon successful registration, the Orchestrator issues a persistent **JWT (JSON Web Token)** to the agent.
- **JWT for Subsequent Communication**: All subsequent communication between the Agent and Orchestrator (heartbeats, job requests, result reporting) must be authenticated using this JWT. The JWT should have a reasonable expiration time and be refreshable. It should be signed by the Orchestrator using a strong secret key.
- **Token Revocation**: The Orchestrator must have the capability to revoke an agent's JWT if it is compromised or the agent is decommissioned.

6.2. Command Whitelisting, Sandboxing, and Protection from Arbitrary Code Execution

This is a critical area to prevent malicious or unintended actions on agent machines.

- **Strict Command Whitelisting**: The Orchestrator must maintain a strict whitelist of commands that agents are allowed to execute. Any command not on this list should be rejected. This whitelist should be configurable and managed by an administrator through the Dashboard.
    - **Implementation Hint**: The Orchestrator should send only predefined command IDs or safe, parameterized commands to agents. Agents should map these to their local whitelisted executables.
- **Agent-Side Validation**: The Agent itself must enforce the command whitelist. Even if a malicious command somehow bypasses the Orchestrator's validation, the Agent should refuse to execute it.
- **Sandboxing (Containerization)**: For higher security, especially when executing potentially untrusted or complex tasks, agents should execute commands within isolated environments. Docker containers are an excellent choice for this:
    - **Implementation Hint**: The Agent could spin up a temporary Docker container for each job, execute the whitelisted command inside it, capture the output, and then destroy the container. This provides strong isolation from the host system.

- **Principle of Least Privilege**: Agents should run with the minimum necessary permissions on the host system. Their user account should not have root/administrator privileges unless absolutely necessary for specific whitelisted tasks.

6.3. Safe Data Storage for Logs and Metrics

- **Database Security**: The Orchestrator's database (SQLite/PostgreSQL) must be secured. This includes:
  - **Access Control**: Restricting database access to only the Orchestrator service.
  - **Encryption**: Encrypting sensitive data at rest (e.g., API keys, tokens if stored) and in transit (e.g., using SSL/TLS for database connections).
  - **Regular Backups**: Implementing a robust backup strategy.
- **Metrics Store Security (Prometheus)**: Prometheus typically stores operational metrics, which are less sensitive than application data. However, access to Prometheus and Grafana should be secured:
  - **Authentication/Authorization**: Grafana should be configured with strong authentication (e.g., OAuth, LDAP, or its internal user management) and role-based access control (RBAC).
  - **Network Isolation**: Prometheus and Grafana should ideally be deployed in a private network segment, accessible only through a secure gateway or VPN.
- **Log Management**: Logs can contain sensitive information. They should be:
  - **Centralized**: Collected and stored in a secure, centralized logging system.
  - **Access Controlled**: Access to logs should be restricted based on roles.
  - **Retention Policies**: Implement clear data retention policies to automatically purge old logs.

7. Scalability Roadmap

To ensure MAMOS can grow beyond its MVP stage, a clear scalability roadmap is essential. This involves planning for increased load, more agents, and potentially supporting multiple organizations.

7.1. Scaling the Orchestrator

- **Horizontal Scaling**: The NestJS Orchestrator can be scaled horizontally by running multiple instances behind a load balancer. This requires the Orchestrator to be stateless (or manage state externally, e.g., in Redis or a shared database).

- **Database Scaling**: Migrating from SQLite to PostgreSQL is the first step for database scalability. For very high loads, consider managed database services (e.g., AWS RDS, Google Cloud SQL) or sharding strategies.
- **Message Queue**: Utilizing Redis as a robust message broker (e.g., with BullMQ or similar libraries) will allow for efficient job queuing and distribution across multiple Orchestrator instances.

7.2. Scaling Agents

- **Agent Deployment**: Agents are inherently distributed. Scaling involves deploying more agents on more machines. The one-click installers simplify this process.
- **Agent Groups/Tenancy**: For supporting multiple organizations or distinct groups of agents, the Orchestrator can be extended to manage agent groups, ensuring agents only receive tasks relevant to their assigned group/organization.

7.3. Scaling Monitoring

- **Prometheus Federation/Thanos**: For large-scale deployments, a single Prometheus instance might not suffice. Prometheus Federation allows multiple Prometheus servers to scrape a central one. Thanos can be used for long-term storage and global query views across multiple Prometheus instances.
- **Grafana Scaling**: Grafana itself is relatively easy to scale horizontally, as it primarily serves dashboards and queries data sources.

7.4. Migration to Kubernetes or Microservices Architecture

- **Container Orchestration**: As the system grows, migrating from Docker Compose to **Kubernetes** (K8s) is a natural progression. Kubernetes provides advanced features for container orchestration, auto-scaling, self-healing, and service discovery.
    - **Implementation Hint**: Each MAMOS component (Orchestrator, Agent, Prometheus, Grafana) can be deployed as a separate microservice within Kubernetes, managed by Deployments, Services, and Ingress controllers.
- **Service Mesh**: For complex microservices interactions, a service mesh (e.g., Istio, Linkerd) can be introduced to handle traffic management, security, and observability between services.
- **Multi-Tenancy**: For supporting multiple organizations, a multi-tenant architecture can be implemented, either by isolating data per tenant (e.g., separate databases or schemas) or by logically partitioning data within a shared database. Kubernetes can facilitate tenant isolation at the infrastructure level.

References

[1] NestJS. (n.d.). *Documentation*. Retrieved from https://docs.nestjs.com/

[2] Prisma. (n.d.). *Documentation*. Retrieved from https://www.prisma.io/docs

[3] SQLite. (n.d.). *SQLite Home Page*. Retrieved from https://sqlite.org/

[4] PostgreSQL. (n.d.). *PostgreSQL: The World's Most Advanced Open Source Relational Database*. Retrieved from https://www.postgresql.org/

[5] Redis. (n.d.). *Redis Labs*. Retrieved from https://redis.io/

[6] Python. (n.d.). *Welcome to Python.org*. Retrieved from https://www.python.org/

[7] Typer. (n.d.). *Typer - Build great CLIs. Fast to develop. Easy to use.* Retrieved from https://typer.tiangolo.com/

[8] Click. (n.d.). *Click — Click Documentation (8.1.x)*. Retrieved from https://click.palletsprojects.com/

[9] Requests. (n.d.). *Requests: HTTP for Humans™. Retrieved from https://requests.readthedocs.io/

[10] websockets. (n.d.). *websockets — websockets 12.0 documentation*. Retrieved from https://websockets.readthedocs.io/

[11] Next.js. (n.d.). *Next.js by Vercel*. Retrieved from https://nextjs.org/

[12] Prometheus. (n.d.). *Prometheus - Monitoring system & time series database*. Retrieved from https://prometheus.io/

[13] Grafana. (n.d.). *Grafana - The open and composable observability platform*. Retrieved from https://grafana.com/

[14] Docker. (n.d.). *Docker: Accelerated Container Application Development*. Retrieved from https://www.docker.com/

[15] Docker Compose. (n.d.). *Overview of Docker Compose*. Retrieved from https://docs.docker.com/compose/

[16] GitHub Actions. (n.d.). *About GitHub Actions*. Retrieved from https://docs.github.com/en/actions

[17] JWT. (n.d.). *JSON Web Tokens - jwt.io*. Retrieved from https://jwt.io/

[18] Kubernetes. (n.d.). *Kubernetes*. Retrieved from https://kubernetes.io/

[19] Istio. (n.d.). *Istio - An open platform to connect, secure, control, and observe microservices*. Retrieved from https://istio.io/

[20] Linkerd. (n.d.). *Linkerd - The ultralight, security-first service mesh for Kubernetes*. Retrieved from https://linkerd.io/