### **Project Restart & Context Load: Zaki-OS**

**Preamble:** This document serves as a complete context load for the ongoing development of the Zaki-OS project. The goal is to re-establish our session with all necessary strategic, architectural, and technical details to continue our work efficiently.

### **Part 1: The Strategic Context**

#### **1.1. Vision, Mission, and Core Values**

* **Vision:** To create a world where human intellect and artificial intelligence converge into a seamless, symbiotic ecosystem, building self-evolving digital organizations that solve humanity's most complex challenges.
* **Mission:** To build Zaki-OS: a decentralized, transparent, and ethically-grounded operating system for intelligence that empowers users to design, orchestrate, and govern collaborative multi-agent systems.
* **Core Values:** The system is guided by principles of Integrity, Human-AI Symbiosis, Continuous Evolution, Principled Modularity, Knowledge as Foundation, and Intentional Design.

#### **1.2. System Architecture**

* **Layered Microservices:** The architecture is split into two primary Docker Compose stacks to separate concerns:
  + **/platform**: For stable, foundational infrastructure (databases, SSO, message brokers, observability tools).
  + **/services**: For custom, rapidly evolving application services (Agent Studio, API Gateway, KMS, etc.).
* **Unified Networking:** All containers communicate over a single external Docker bridge network named zaki-network.
* **SSO & Routing:** We are implementing a centralized authentication system using **Traefik** as a reverse proxy and **Authelia** as the Single Sign-On (SSO) provider. The goal is **path-based routing** from a single domain (zakios.hasan-nuseibeh.com), not subdomain-based routing.
* **Service Discovery:** We are in the process of adding **Consul** to the platform stack to enable dynamic discovery of services, which will power a future Command and Control Center.

#### **1.3. Current Project State**

* **Platform Stack is Running:** We have successfully consolidated most infrastructure components into a single platform/docker-compose.yml file. All 15 platform containers (Postgres, Redis, Traefik, Authelia, Consul, etc.) are running.
* **Code Sourced from Backups:** The project is a **consolidation effort**. Authoritative code for rich, feature-complete services like Agent Studio, KMS, and Resource Center has been located in project backups (backups/eidolon\_restored/) and is being integrated into the new, clean architecture.
* **Critical Issue - Agent Chaos:** A major motivation for the current work is that uncoordinated AI agents (from Cursor, Windsurf) have been making conflicting changes to the Docker infrastructure, causing instability. This highlights the need for a governed **Agent Operations Center**.
* **Immediate Blocker - Authelia Login Loop:** While the platform services are running, the SSO system is not fully functional. Users attempting to log in via Authelia are stuck in a redirect loop.

### **Part 2: The Technical State**

#### **2.1. Authoritative platform/docker-compose.yml**

This is the current, consolidated configuration for all infrastructure services.

# platform/docker-compose.yml  
version: '3.8'  
services:  
 # SSO & PROXY  
 traefik:  
 image: traefik:v2.11  
 container\_name: zaki-traefik  
 # ...ports, volumes, labels...  
 authelia:  
 image: authelia/authelia:4  
 container\_name: zaki-authelia  
 # ...volumes, labels for path-based routing...  
 # CORE INFRASTRUCTURE  
 postgres:  
 image: postgres:15-alpine  
 container\_name: zaki-postgres  
 # ...config...  
 redis:  
 image: redis:7-alpine  
 container\_name: zaki-redis  
 # ...config...  
 qdrant:  
 image: qdrant/qdrant:latest  
 container\_name: zaki-qdrant  
 # ...config and labels for path-based routing...  
 rabbitmq:  
 image: rabbitmq:3-management-alpine  
 container\_name: zaki-rabbitmq  
 # ...config and labels for path-based routing...  
 # PLATFORM SERVICES  
 consul:  
 image: consul:1.15.4  
 container\_name: zaki-consul  
 # ...config and labels for path-based routing...  
 elasticsearch:  
 image: elasticsearch:8.11.0  
 container\_name: zaki-elasticsearch  
 # ...config...  
 logstash:  
 image: logstash:8.11.0  
 container\_name: zaki-logstash  
 # ...config...  
 kibana:  
 image: kibana:8.11.0  
 container\_name: zaki-kibana  
 # ...config and labels for path-based routing...  
 prometheus:  
 image: prom/prometheus:latest  
 container\_name: zaki-prometheus  
 # ...config and labels for path-based routing...  
 grafana:  
 image: grafana/grafana:latest  
 container\_name: zaki-grafana  
 # ...config and labels for path-based routing...  
 n8n:  
 image: n8nio/n8n:latest  
 container\_name: zaki-n8n  
 # ...config and labels for path-based routing...  
 ollama:  
 image: ollama/ollama:latest  
 container\_name: zaki-ollama  
 # ...config...  
 docs-site:  
 build: ../docs-site  
 container\_name: zaki-docs  
 # ...config and labels for path-based routing...  
volumes:  
 # ...all necessary named volumes...  
networks:  
 zaki-network:  
 external: true

#### **2.2. Service Inventory Summary**

* **Agent Studio:** The authoritative version exists in the backups. It's a feature-rich FastAPI service for managing the full lifecycle of agents, including configuration, memory, testing, sessions, snapshots, and usage/cost tracking.
* **KMS (Knowledge Management System):** A YAML-driven system backed by Postgres and Qdrant. It serves as the central brain for memory, values, and decision logging.
* **Resource Center:** An intelligent service for dynamically provisioning and managing all tools, APIs, and LLM models available to agents, complete with SLA scoring and governance checks.
* **Plane:** A full-featured project management system (like Jira) that will be leveraged for the **Agent Operations Center** to track tasks assigned to AI agents.
* **C2 Center (Command & Control):** A planned unified dashboard for monitoring and managing the entire Zaki-OS stack. It will be powered by data from Consul and other observability tools.

#### **2.3. Authelia Login Loop - Technical Diagnosis**

The immediate issue preventing progress is the SSO login loop. The suspected causes are:

1. **Cookie Domain Mismatch:** The Authelia configuration may not be correctly set up to handle sessions for both localhost (for development) and the production domain (zakios.hasan-nuseibeh.com).
2. **Password Hash Issue:** The user database (users\_database.yml) may be using an outdated Bcrypt hash, while the latest version of Authelia expects an Argon2id hash.
3. **Traefik Middleware Configuration:** The forward-auth redirect URLs in the Traefik labels need to be correctly configured for both the localhost and production routing rules.

### **Part 3: The Immediate Task**

**Your mission is to resolve the "Authelia Login Loop."**

This requires you to generate a set of corrected configuration files that address all three suspected causes. Your deliverable should be a set of code blocks containing the updated contents for:

1. **platform/authelia/configuration.yml**: It must be updated to handle session cookies for both the localhost and zakios.hasan-nuseibeh.com domains.
2. **platform/authelia/users\_database.yml**: It should be updated with a placeholder for a new **Argon2id** password hash, along with a command showing how to generate one.
3. **platform/docker-compose.yml**: The Traefik labels for the authelia and grafana services must be updated to define and use separate middleware configurations for local and production access, ensuring the correct redirect URLs are used in each case.

Once these files are updated, the user will be able to restart the platform and have a fully functional SSO system for both local development and production access.

### Part 4: Detailed Implementation Blueprints

This section provides more granular, actionable plans for building out the core application services.

#### 4.1. Agent Studio Backend

* **Purpose:** To serve as the central hub for creating, managing, testing, and monitoring all AI agents.
* **Backend API Requirements:**
  + **Models:** Must have SQLAlchemy models for Agent, Session, UsageLog, AgentSnapshot, and EvaluationRun, each with appropriate fields (e.g., JSONB for conversation history, Numeric for cost).
  + **Core CRUD:** Implement full POST, GET, PUT, DELETE endpoints for /agents/.
  + **LLM Integration:** Provide endpoints like GET /models/providers and GET /models/{provider}/list that can dynamically query the **Ollama** service to discover available models.
  + **Memory Management:** Implement endpoints like POST /agents/{agent\_id}/memory/add which acts as a proxy to the **KMS Service** for document ingestion, and POST /agents/{agent\_id}/memory/query which proxies semantic search requests to the KMS.
  + **Testing & Execution:** Implement a POST /agents/{agent\_id}/test/chat endpoint that streams token-by-token responses from the assigned LLM. A POST /agents/{agent\_id}/code/execute endpoint must proxy requests to the **Coding Center**.
  + **Usage Tracking:** Every endpoint that results in an LLM call must create a record in the UsageLog table, capturing token counts and calculated cost.
* **Frontend UI Integration:** The API must provide all necessary data to power the multi-tabbed UI seen in screenshots, including agent configuration, memory sources, test chat, session history, and usage analytics.

#### 4.2. KMS (Knowledge Management System) Backend

* **Purpose:** To function as the system's long-term memory, value system, and auditable log.
* **Backend API Requirements:**
  + **Document Pipeline:** Implement a POST /documents/upload endpoint that manages the full RAG ingestion pipeline: document parsing, text chunking, vector embedding (by calling the **Resource Center** for an embedding model), and storage into **PostgreSQL** (metadata) and **Qdrant** (vectors).
  + **Search API:** Implement a POST /search/semantic endpoint that takes a natural language query, embeds it, and returns the most relevant text chunks from the knowledge base.
  + **Governance API:** Implement endpoints like GET /values and GET /decisions to serve the system's core principles and decision logs from their respective YAML files or database tables.

#### 4.3. Resource Center Backend

* **Purpose:** To abstract and govern the use of all tools, APIs, and LLM models.
* **Backend API Requirements:**
  + **Configuration:** The service must parse a resources.yaml file on startup, which declaratively defines all available resources (Docker tools, external APIs, Ollama models).
  + **Core Smart Router:** The central endpoint, POST /request-capability, must be implemented. Its logic will:
    1. Find resources matching the requested capability.
    2. Query the **KMS** to perform a governance check (e.g., budget, policy).
    3. Select the optimal resource based on SLA scores.
    4. Proxy the request to the chosen resource (e.g., an Ollama model or an external API), securely injecting any necessary API keys.
    5. Log the transaction in a ResourceUsageLog table and update the resource's SLA score.

#### 4.4. Agent Operations Center (AOC) Backend

* **Purpose:** To act as the central task manager and governance hub for external AI agents (from Cursor, Windsurf, etc.).
* **Backend API Requirements:**
  + **Integration with Plane:** This service will act as a bridge to the existing Plane project management system.
  + **Task Check-in/Check-out:**
    - POST /tasks/check-in: An agent calls this to request permission to work on a task. The backend validates the request, checks for resource conflicts (by seeing if files are "locked" by another task), and if approved, creates a new "Issue" in Plane, effectively locking the resources.
    - POST /tasks/{task\_id}/check-out: An agent calls this upon completion. The backend adds the agent's final report as a "Comment" on the Plane Issue and releases the resource locks.
  + **Dashboard API:** A GET /tasks endpoint will query the Plane API to fetch the status of all ongoing agent tasks to display in the C2 Center UI.

### Part 5: Advanced System Blueprints & Governance

This section details the implementation plans for higher-level management and control systems that build upon the core services.

#### 5.1. Command & Control (C2) Center Backend

* **Purpose:** To consolidate the various scattered dashboards into a single, unified, and dynamic interface for monitoring and managing the entire Zaki-OS stack. Its primary mandate is to provide a "single pane of glass" for system operators.
* **Architectural Approach:** The C2 Center will not have a static list of services. Instead, it will dynamically discover and display services by querying the **Consul** service registry.
* **Backend API Requirements:**
  + **GET /system/status**: This is the core dynamic endpoint. Its logic must:
    1. Make an API call to the zaki-consul service at /v1/catalog/services to get a list of all registered services.
    2. For each service, retrieve its tags, which will contain custom zaki-os.\* metadata (e.g., category, UI path, description).
    3. Concurrently, query the Docker Engine API to fetch the live operational status (running, unhealthy), resource usage (CPU, Memory), and other runtime metadata for each container.
    4. Aggregate this information into a single JSON response that dynamically drives the entire C2 frontend, ensuring the dashboard always reflects the true state of the system.
  + **POST /containers/{id}/{action}**: A unified endpoint for container lifecycle management (e.g., actions like start, stop, restart). This API will execute the corresponding commands via the Docker SDK for Python, providing a secure bridge for the UI to interact with the Docker daemon.
  + **WS /ws/system/stream**: A WebSocket endpoint that provides a continuous stream of system-wide events, including health status changes and resource metric updates, allowing the UI to update in real-time without polling.

#### 5.2. Governance & Agent Operations Protocol (AOC)

* **Purpose:** To solve the "agent chaos" problem by establishing a formal governance protocol that all external agents (Cursor, Windsurf, etc.) must adhere to before modifying system resources.
* **Architectural Approach:** This protocol is implemented by the **Agent Operations Center (AOC)**, which leverages the existing **Plane** service as its system of record.
* **Detailed POST /tasks/check-in Logic:**
  + **Resource Manifest:** The request from an agent must include a resource\_manifest array, explicitly listing the files or services it intends to modify (e.g., ["services/agent-studio/app/main.py", "platform/docker-compose.yml"]).
  + **Conflict Resolution:** Upon receiving a check-in request, the AOC backend queries the Plane API to see if any currently "In Progress" Issues have a lock on any of the resources in the manifest.
  + **Locking Mechanism:** If there are no conflicts, the AOC creates a new Issue in Plane, populates its description with the agent's plan, and crucially, saves the resource\_manifest to a custom field. This Issue now represents an **active lock** on those resources. The agent receives the Issue ID as its task\_id and is authorized to proceed.
  + **Rejection:** If a resource is already locked, the AOC returns a 423 Locked error to the requesting agent, providing the task\_id of the conflicting task, allowing the agent to wait or report the conflict.
* **Detailed POST /tasks/{task\_id}/check-out Logic:**
  + Upon successful completion of its work, the agent calls this endpoint with its final report.
  + The AOC backend appends the report as a comment to the Plane Issue, updates the Issue's status to "Done," and—most importantly—**clears the resource lock field**, making the resources available for the next agent.