# **Project Name:** Cloud Odyssey
# Design Document
# **Date:** 04/04/2025
# **Version:** 1.0

---



| | |
|---|---|
| 21116034 | Vaishnavi Virat Dave |
| 21116048 | Kaashvi Jain |
| 23114043 | Kajal |
| 23114052 | Kumud |
| 21323003 | Abhijna Raghavendra |
| 23114008 | Anushka Jangid |



---

## 1. Summary


Cloud Odyssey is a **distributed computing system** designed using a **microservices architecture** to ensure modularity, scalability, and ease of deployment. The system enables users to efficiently schedule, execute, and monitor tasks across multiple worker nodes in a cluster.


## 2. Main Design

### High-Level Design (HLD)

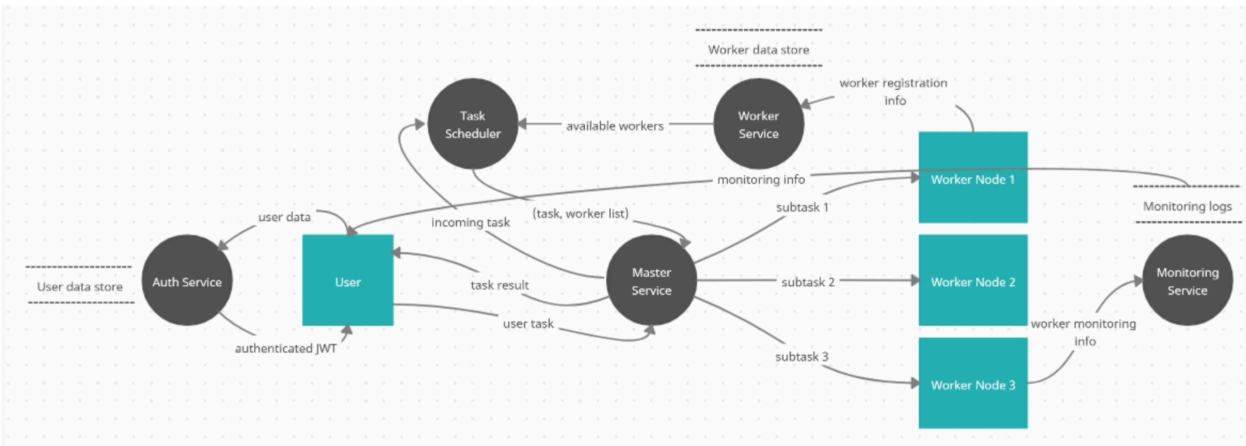
Each microservice is designed with specific functionalities and exposes REST API endpoints for interaction.

**- Microservices Overview:**

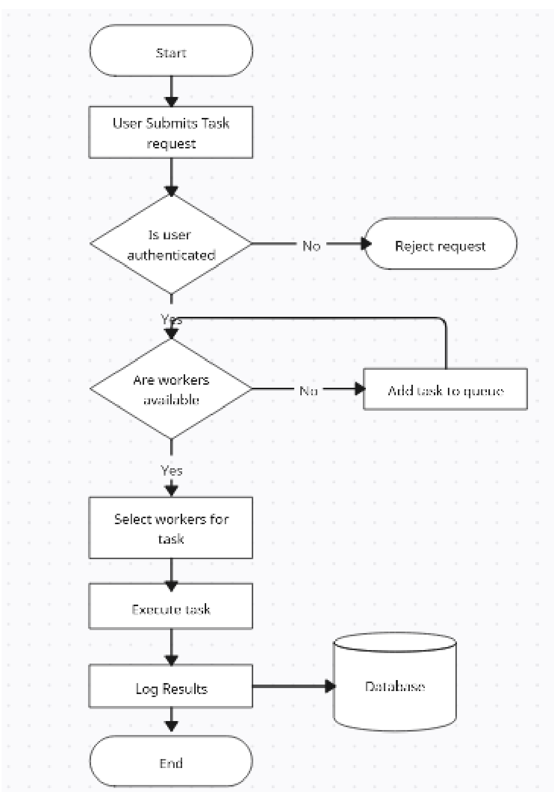
1. **Authentication Service:** Handles user authentication and authorization using JWT.
2. **Task Scheduler Service:** Assigns tasks to available workers based on system specifications and load.

3. **Worker Service:** Manages the registration and health status of worker nodes.
4. **Master Service:** Executes tasks on selected worker nodes via SSH and reports results.
5. **Monitoring Service:** Collects real-time system metrics and provides a dashboard for analysis.

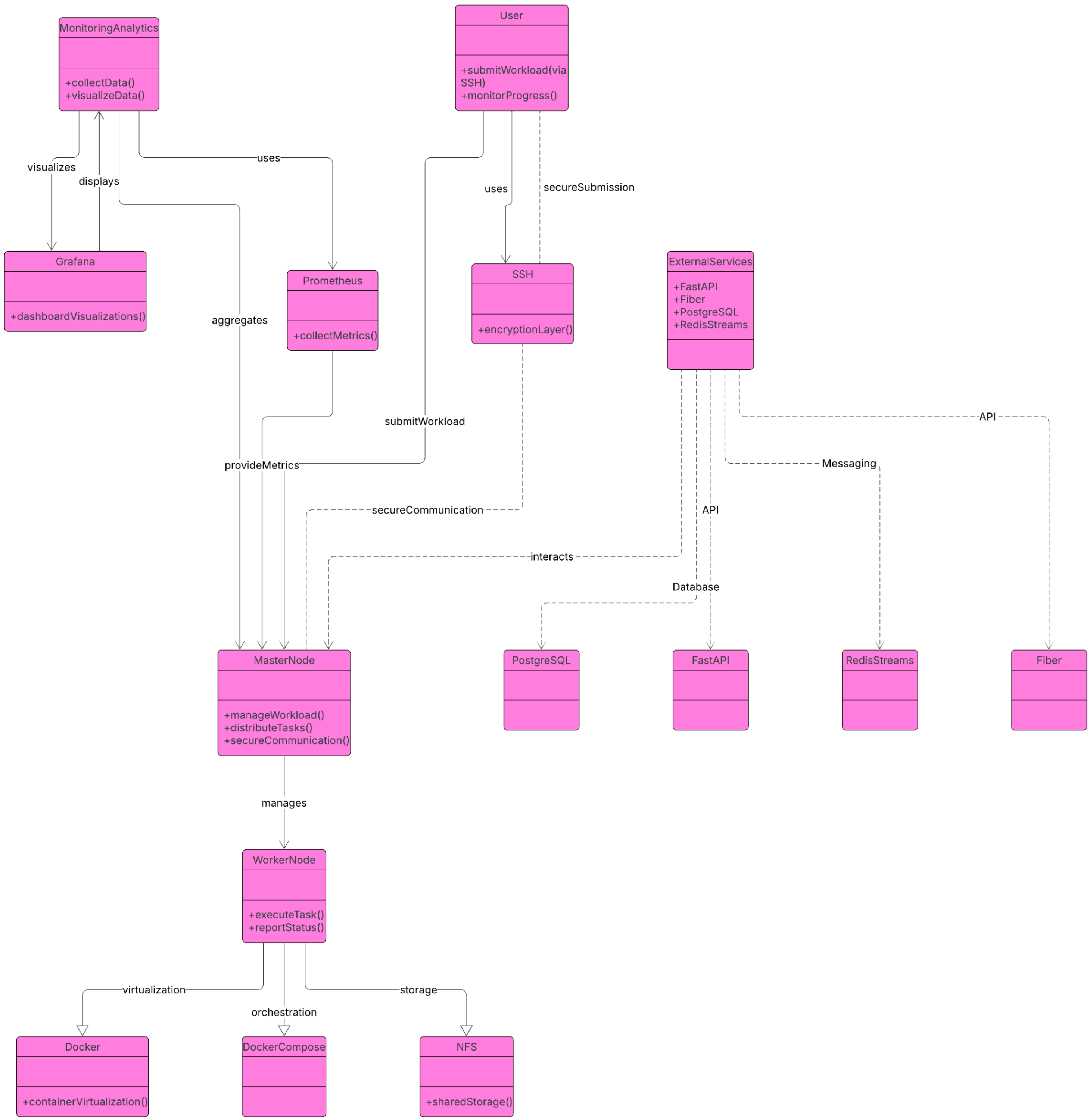
### - Data Flow Diagrams (DFD)




### - Flowchart: Task Execution



# UML Diagram(Class Diagram)



**Low-Level Design (LLD)**

- **Microservices API Specification**

### a. Authentication Service

**Handles user authentication and authorization using JWT.**

- `POST/auth/register` **– Register a new user**
- `POST/auth/login` **– Authenticate a user and issue a token**
- `GET/auth/user` **– Get authenticated user details**
- `POST/auth/logout` **– Invalidate the user's session**

---

### b. Task Scheduler Service

**Assigns tasks to available workers based on system specifications and load.**

- `POST/scheduler/task` **– Submit a new task for scheduling**
- `GET/scheduler/task/{task_id}` **– Retrieve the status of a scheduled task**
- `GET/scheduler/workers` **– Get the list of available workers**
- `DELETE/scheduler/task/{task_id}` **– Cancel a scheduled task**

---

### c. Worker Service

**Manages the registration and health status of worker nodes.**

- `POST/worker/register` **– Register a new worker node**
- `GET/worker/health/{worker_id}` **– Check the health status of a worker**
- `DELETE/worker/{worker_id}` **– Deregister a worker node**
- `GET/worker/list` **– Get all registered workers**

---

### d. Master Service

**Executes tasks on selected worker nodes via SSH and reports results.**

- `POST/master/execute` **– Execute a command on a worker node**
- `GET/master/result/{task_id}` **– Fetch the result of an executed task**
- `DELETE/master/task/{task_id}` **– Stop a running task**

---

### e. Monitoring Service

**Collects real-time system metrics and provides a dashboard for analysis.**

- `GET/monitoring/metrics` **– Fetch system metrics (CPU, memory, etc.)**
- `GET/monitoring/logs/{worker_id}` **– Retrieve logs for a specific worker**
- `POST/monitoring/alert` **– Create an alert based on system thresholds**
- `GET/monitoring/alerts` **– Get active system alerts**

---

- **Task Scheduling Algorithms:**
    1. **Round Robin:** Distributes tasks in a cyclic manner among available worker nodes to ensure fairness. This prevents any single node from being overloaded while maintaining a consistent distribution of tasks.
    2. **Shortest Job First (SJF):** Prioritizes tasks based on estimated execution time, ensuring that smaller tasks complete first to minimize overall task completion time and reduce queue wait times.
    3. **Weighted Scheduling:** Implements an advanced task allocation strategy by assigning dynamic weights to tasks based on multiple parameters such as:
        a. Input size: Larger input files may require additional computation and should be allocated accordingly.
        b. Program complexity: Tasks with higher computational complexity should be distributed to more capable nodes.
        c. Expected execution time: Predicting execution times helps in balancing loads efficiently across the cluster.

d. System load: Real-time system metrics are utilized to adjust scheduling dynamically, ensuring optimal resource utilization without overwhelming specific nodes.

A basic mathematical model for task weight assignment can be defined as:

$\mathbf{W = \alpha S + \beta C + \gamma T + \delta L}$, where W□ is the weight assigned to the task, S is the size factor, C represents the program complexity metric, T is the expected execution time, L is the current system load metric, α, β, γ, δ are tuning coefficients that adjust the influence of each parameter on scheduling decisions.

The scheduler dynamically updates these weights based on real-time cluster metrics and historical data to optimize task allocation and execution efficiency.

- **Deployment Strategy**
  - **Docker**: Each microservice is containerized for consistency across environments.
  - **Kubernetes**: Manages deployments with auto-scaling, service discovery, and persistent storage.
  - **CI/CD (GitHub Actions)**: Automates build, testing, and deployment to a container registry.
  - **Rollout Strategies**: Uses rolling updates, blue-green, or canary deployments for minimal downtime.

# 3. Questions

a. How should authentication and authorization be managed across services?
b. What fallback mechanism should be in place if a worker node fails mid-task?
c. Should the system prioritize high-performance nodes or focus on balanced workload distribution?
d. How frequently should worker nodes send health updates?
e. What is the best way to handle failures in remote SSH execution—retry, reassign, or fail?
f. Should task scheduling be dynamic based on system load, or should it follow predefined rules?
g. How should monitoring and logging be structured for maximum visibility?
h. What should be the rollback strategy in case of failed deployments?

i. Should CI/CD pipelines include automated testing before deployment?
j. How can security be enforced in worker registration and task execution?

## 4. Comments

**Authentication and Authorization Management:** The system uses JWT-based authentication, which is appropriate for microservices. However, implementing OAuth2 with a centralized identity provider (such as Keycloak) could improve security and user management.

**Fallback Mechanism for Worker Node Failures:** If a worker node fails mid-task, a re-execution policy should be in place. The scheduler should monitor worker health and reassign failed tasks dynamically to another available node.

**Task Scheduling Strategy:** We suggest weighted scheduling, which is beneficial. However, it should be configurable to support both high-performance prioritization and balanced workload distribution based on real-time metrics.

**Worker Health Updates Frequency:** This should be dynamic based on load conditions. A push-based system with an event-driven health check mechanism (e.g., using WebSockets) may provide better efficiency.

**Handling Failures in Remote SSH Execution:** Retrying with exponential backoff is a good strategy. If failure persists, the system should log the failure and reassign the task to another node.

**Dynamic vs. Rule-Based Scheduling:** A hybrid approach can be used where predefined rules exist, but the scheduler dynamically adjusts priorities based on real-time system performance.

**Monitoring and Logging Structure:** The monitoring service fetches system metrics and logs, but integrating a centralized log aggregation tool like Prometheus with Grafana will improve visibility.

**Rollback Strategy for Failed Deployments:** Blue-green deployments seem to be the safest approach, allowing rollback to the last stable version with minimal downtime.

**CI/CD Testing Before Deployment:** Automated testing should be enforced as part of the pipeline, including unit tests, integration tests, and security scans before deployment.

**Security in Worker Registration and Task Execution:** Secure worker registration should use mutual TLS authentication, and task execution should be sandboxed to prevent unauthorized system access.