# Project Requirements: ORC AI Workflow Orchestration System

## 1. Functional Requirements

- **Workflow Orchestration:** Provide robust DAG-based orchestration using Apache Airflow. Users can define **"pipelines as code"** in Python with dynamic DAG generation and parameterizationairflow.apache.orgairflow.apache.org. Each workflow (DAG) contains tasks (operators) with explicit dependencies, supporting parallel execution, retries, conditional branching, and triggering on schedules or events. The system should support custom operators, sensors, and hooks to connect to external services or data sourceskomodor.com.
- **Task Execution & Integration:** Execute arbitrary tasks on distributed workers. Support Celery or Kubernetes executors to run tasks concurrently. Integrate other orchestration tools: e.g., invoke Prefect flows and Dask jobs as part of DAGs. Prefect (open-source Python tool) can be used for complex data/ML workflows, giving an alternate orchestration layerdatacamp.com. Use Dask's parallel-computing library for heavy data-processing tasks, enabling large-scale computation across a cluster.
- **Dynamic Pipelines:** Support dynamic DAG creation and parameterization at runtime. Pipelines should be configurable via variables or external inputs, allowing adaptation to changing data or context.
- **Monitoring and Metrics:** Collect real-time metrics from the Airflow cluster (scheduler, workers, tasks) and underlying resources. Use StatsD exporters and Prometheus for metric collectionredhat.com. Provide Grafana dashboards for key indicators (task durations, success/failure rates, queue depths, resource usage). Offer centralized logging of task output and audit logs for all operations.
- **Alerting and Notifications:** Implement alerting on failures, SLA breaches, or resource issues. On critical events, send immediate notifications to Slack (e.g., via Airflow's `SlackAPIPostOperator`) or emailmedium.com. Integrate Prometheus Alertmanager with Slack webhooks for automated alerts when metrics exceed thresholds. Alerts should include contextual information (DAG name, task, error) and provide links back to the UI/logs.
- **Failure Prediction (AI/ML):** Analyze historical execution data using machine learning to predict likely failures or delays before they occur. The system should proactively flag pipelines at risk and suggest mitigations. As IBM notes, AI/ML can "predict failures and optimize workflows dynamically" by learning patterns from workflow dataibm.com. For example, if certain tasks often fail under specific conditions, ORC AI should highlight this and perhaps trigger corrective actions (e.g. increased retries or resource allocation).
- **Knowledge Graph Queries:** Use Neo4j to store metadata about workflows, tasks, data assets, and dependencies. Support rich Cypher queries so users can explore the **knowledge graph** of pipelines. For example, users can find all downstream tasks connected to a given

data source or identify which pipelines would be impacted by a change. Neo4j's property graph will enable flexible lineage and impact analysis.

- **User Interface and Visualization:** Provide a user-friendly web UI. Leverage Airflow's built-in UI (tree/graph views, Gantt charts, logs viewer) plus custom dashboards. For specialized visualization or reports, use Streamlit (an open-source Python app frameworkstreamlit.io) to build interactive dashboards or wizards (e.g. for querying the knowledge graph or viewing AI predictions).
- **Logging and Auditing:** Maintain detailed logs for every task execution and user action. Logs should be easily searchable (e.g., via the Airflow UI or log aggregation). Track who triggered what and when, for compliance. Store logs centrally (e.g., in an ELK/EFK stack or object store) for long-term retention and analysis.
- **Versioning and CI/CD:** Integrate with Git (or other VCS) for DAG and pipeline code. Support automated testing of workflows (unit tests, integration tests). Enable CI/CD pipelines to validate and deploy new DAGs and ML models into Airflow. Maintain versioned archives of workflow definitions to allow rollback.
- **Role-Based Access Control:** Enforce permissions so users only access data and features appropriate to their role. Airflow's RBAC (via Flask AppBuilder) or an equivalent system should restrict who can see or edit which DAGs and data.

# 2. Non-Functional Requirements

- **Performance:** Schedule and kick off tasks with low latency (e.g., scheduling delays < few seconds). Support high throughput (hundreds of tasks started per minute) without bottlenecks. UI and dashboard refreshes should be snappy (data updated every few seconds). Keep monitoring metric scrape intervals frequent (e.g., 10–15s).
- **Scalability:** Architect for horizontal scaling. Airflow components (Scheduler, Webserver, Workers) should be deployable on multiple nodes. The Celery/Kubernetes executor must allow adding workers to increase throughput. Dask clusters should auto-scale to match parallel task demand. Prometheus and Grafana must handle growing volumes of metrics and dashboards. As Astronomer notes, Airflow's "scalable and resilient architecture" can handle increasing workloads without losing reliabilityastronomer.io.
- **Availability:** Design for high availability (HA). Critical services (PostgreSQL metadata DB, Redis/RabbitMQ broker, Neo4j) must run in HA mode (e.g. multi-AZ clusters, replication). Use redundant Airflow schedulers and webservers (behind a load balancer) so no single point of failure. Ensure zero-downtime deploys when updating components. Automatic retries and fallback actions should mitigate transient failures.
- **Security:** Secure the system end-to-end. Encrypt all communication (TLS for web UI, API, DB connections). Store secrets (DB passwords, Slack tokens) securely (e.g. using Vault or encrypted configs). Apply principle of least privilege: each component (Airflow, Neo4j, Prometheus, etc.) runs under restricted accounts. Use authenticated, role-based access (e.g., OAuth/LDAP for Airflow) and audit all access. Regularly patch and update all software to address vulnerabilities.

- **Maintainability:** Keep the system modular and well-documented. Use containerization (Docker) and orchestration (Kubernetes) to isolate components and simplify deployments. Provide clear logging and health metrics to ease troubleshooting. Write automated tests for workflows and monitoring, and use CI pipelines for deployment. Adopt configuration management (Helm charts, Terraform) to version infrastructure.
- **Usability:** The interfaces must be intuitive. The Airflow UI should be accessible and clearly organized. Custom dashboards (e.g. Streamlit) should present complex data simply. Provide adequate documentation and examples for developers. Ensure alerts are readable and actionable. For non-technical stakeholders, offer high-level overviews of system status and pipeline results.

# 3. System Architecture & Integration Overview

The ORC AI system is built around Apache Airflow as the core orchestrator. Figure 1 illustrates a simplified monitoring integration:

*Figure 1: Airflow components (Webserver, Scheduler, Workers) emit metrics (via StatsD) to a StatsD exporter; Prometheus scrapes these metrics; Grafana visualizes them*[redhat.com](redhat.com).

- **Airflow Core:** The Airflow Scheduler, Webserver, and optional Triggerer run on dedicated nodes. All components use a central PostgreSQL metadata database. DAG files are stored in a shared repository (e.g., Git-synced directory). The Celery executor uses Redis or RabbitMQ as a message broker to dispatch tasks to worker nodes (which may also scale up/down on demand).
- **Monitoring Integration:** Airflow is configured to emit StatsD metrics. A StatsD exporter service receives these metrics and exposes them in Prometheus format[redhat.com](redhat.com). Prometheus (running on its own cluster or in Kubernetes) scrapes the exporter and stores time-series data. Grafana connects to Prometheus to provide dashboards on DAG status, task durations, system load, etc. Alerts are defined in Prometheus (or Alertmanager) for conditions like high failure rate, long queue times, or resource exhaustion.
- **Alerting & Notifications:** For real-time alerts, we integrate Slack in two ways. First, Airflow can send Slack messages directly: e.g., using the `SlackAPIPostOperator` in a task's `on_failure_callback` to notify a Slack channel[medium.com](medium.com). Second, Prometheus Alertmanager can use Slack Webhook integration: when Prometheus rules fire, Alertmanager pushes a notification payload to Slack, reaching responsible teams immediately.
- **Knowledge Graph (Neo4j):** A Neo4j database runs as a separate service. We periodically load Airflow's DAG and task metadata (names, parameters, run history) into Neo4j, either via custom Airflow tasks or CDC pipelines. This creates a graph where nodes represent pipelines, tasks, and data assets, and edges represent dependencies and data flows. Users (via the UI or Streamlit apps) can query this graph with Cypher to answer lineage questions (e.g., "which pipelines consume this database?").
- **Prefect Integration:** Prefect can be invoked from Airflow either by calling the Prefect CLI/API within a task, or by embedding Prefect flows as Python tasks. This allows leveraging Prefect's strengths (such as its dynamic mapping or cloud UI) for some

workflows. Prefect's results can feed back into Airflow or the knowledge graph for continuitydatacamp.com.

- **Dask Cluster:** For tasks requiring heavy parallel computation, Airflow workers can submit jobs to a Dask cluster. This is typically done via the Dask Python client (using gRPC to communicate with the Dask scheduler). Airflow can use a DaskExecutor (if supported) or simply call Dask-using code as part of a PythonOperator. Dask workers run on dedicated compute nodes (possibly auto-scaling on Kubernetes).
- **Streamlit UI:** Custom dashboards and tools are built with Streamlit (v1.45.0). These apps query Airflow's metadata DB or Neo4j using Python drivers/ORMs. For instance, a Streamlit app might let a user select a pipeline and visualize its DAG graph, recent run statuses, or AI-predicted risks. Streamlit provides an interactive UI (charts, tables) without requiring front-end codingstreamlit.io.
- **Communication Protocols:** Most components communicate over standard protocols: Airflow's web UI and API use HTTPS; StatsD metrics use UDP; Prometheus scrapes over HTTP; Slack uses HTTPS; Neo4j uses Bolt or HTTPS for queries; Dask uses gRPC. All internal networks are assumed secure or within a VPC.

# 4. User Roles and Permissions

We define roles to segregate duties and permissions. Airflow's RBAC (Flask AppBuilder) will enforce these rolesairflow.apache.org. A typical mapping might be:

| Role | Capabilities | Access Level |
|---|---|---|
| **Admin** | System configuration, user/role management, high-level oversight. Manages Airflow cluster (upgrade, config), configures integrations (Slack, Neo4j, monitoring), and handles security. | Full system access (deploy/manage Airflow, databases, secrets). |
| **Developer** | Creates and updates workflow definitions (DAG code). Tests and deploys pipelines. Can trigger and inspect DAG runs, view logs. | Write access to code repository; write/edit permissions on specific DAGs; view logs and metrics; cannot change system configs. |
| **Operator** | Monitors pipeline execution (using UIs and dashboards). Responds to failures (retries tasks, fixes issues). Pauses/resumes or reruns DAGs as needed. | Can execute and manage runs of existing DAGs; read access to code; no access to modify pipeline definitions or core configs. |
| **Viewer** | Observes system status and logs. Performs audits or analysis. | Read-only access to dashboards, task logs, and reports; cannot trigger or edit pipelines. |

Airflow explicitly defines similar roles: a *Deployment Manager* (Admin), *DAG author* (Developer), and *Operations user* (Operator)airflow.apache.org. These map directly to our Admin/Developer/Operator roles. Access to Slack channels and Neo4j queries can be further controlled (e.g. private channels or database credentials) to match these roles.

# 5. Technology Stack Specification

| Component | Technology (Version) | Notes / Purpose |
|---|---|---|
| **Orchestration** | Apache Airflow 3.0.1 | Core workflow engine. Python 3.10. CeleryExecutor |

| Component | Technology (Version) | Notes / Purpose |
|---|---|---|
| | (stable) | with Redis broker or KubernetesExecutor. Airflow providers for Slack, StatsD, Neo4j. |
| **Workflow Library** | Prefect 2.x | Secondary orchestration for ML pipelines. Installed via `pip install prefect`. Prefect Core library and CLI. |
| **Parallel Compute** | Dask & dask.distributed 2025.x | Distributed task execution. Dask Scheduler and Workers (Python). Dask==2025.x or latest. |
| **Monitoring** | Prometheus 2.50+, Grafana 10.0+ | Time-series metrics. Prometheus scrapes metrics; Grafana v10 for dashboards. Uses `prometheus_client` Python lib. |
| **Knowledge Graph** | Neo4j 5.x (Enterprise) | Graph DB for metadata. Bolt protocol and Cypher. Neo4j Python Driver (`neo4j-driver`). |
| **Notification** | Slack API v2, Incoming Webhooks | Airflow's `SlackAPIPostOperator` for notificationsmedium.com. Slack apps for Alertmanager integration. |
| **UI Framework** | Streamlit 1.45.0 | Builds interactive dashboards and toolsstreamlit.io. Pure-Python frontend. |
| **Database** | PostgreSQL 14 | Airflow metadata DB. Setup in HA (replication). SSL enabled. |
| **Broker** | Redis 7.x or RabbitMQ 3.x | Message broker for CeleryExecutor. HA setup (Redis Sentinel or RabbitMQ cluster). |
| **Python** | 3.10+ | Runtime for all components (Airflow, Prefect, Dask, Streamlit). |
| **Infrastructure** | Linux (Ubuntu 22.04 LTS) / Kubernetes 1.25+ | Host OS or container platform. Docker 20.x for images. K8s for deployment, autoscaling. |
| **CI/CD & Dev** | Git, Helm, Terraform | Version control and deployment automation. Linting/testing tools (pytest, mypy, etc.). |

All Python components share a compatible environment (3.10) and use virtual environments or containers. Key libraries include `apache-airflow[statsd,slack,neo4j]`, `prefect`, `dask[distributed]`, `prometheus_client`, `neo4j-driver`, and `slack-sdk`. The system will be deployed on hardened Linux servers or containers behind firewalls/VPCs. Regular dependency updates and patching are required for security.

This requirements document outlines the full scope of ORC AI's functionality, quality attributes, architecture, user roles, and chosen technologies. It serves as a roadmap for development and a reference for stakeholders, ensuring that the system meets its goals of scalable, reliable, and intelligent workflow orchestration.