# ORC AI Prototype Development Plan

## Phase 1: Environment Setup

* **Key deliverables:** Install Apache Airflow in Docker (via docker-compose), configure Prometheus and Grafana, enable Slack notifications, and have Neo4j Desktop running locally. The Airflow containers include the Scheduler, Webserver, and database. Prometheus and Grafana run as separate containers to scrape and display metrics. Slack integration is set up so that Airflow or monitoring alerts can post to a channel. Neo4j Desktop provides a local graph DB for the prototype (it’s “designed to help you… learn and experiment with Neo4j locally”[neo4j.com](https://neo4j.com/docs/desktop-manual/current/#:~:text=Neo4j%20Desktop%20is%20a%20client,the%20resources%20of%20your%20machine)).
* **Tools/Libraries:** Docker & Docker Compose, official Airflow Docker image (e.g. apache/airflow), PostgreSQL (or SQLite) for Airflow’s metadata DB, Prometheus and Grafana Docker images, StatsD exporter for metric bridging, the apache-airflow-providers-slack package for SlackOperator (configured via an Airflow Slack connection[airflow.apache.org](https://airflow.apache.org/docs/apache-airflow-providers-slack/stable/notifications/slack_notifier_howto_guide.html#:~:text=BashOperator%28%20task_id%3D,general%22%2C%20username%3D%22Airflow%22%2C%20%29)), and Neo4j Desktop for the graph.
* **Estimated time:** 1–3 days.
* **Example:** A docker-compose.yml might include services like:

|  |
| --- |
| services:  postgres:  image: postgres:13  environment:  POSTGRES\_USER: airflow  POSTGRES\_PASSWORD: airflow  redis:  image: redis:latest  airflow-webserver:  image: apache/airflow:2.6.1  restart: always  depends\_on:  - postgres  - redis  environment:  AIRFLOW\_\_CORE\_\_EXECUTOR: LocalExecutor  AIRFLOW\_\_CORE\_\_SQL\_ALCHEMY\_CONN: postgresql+psycopg2://airflow:airflow@postgres/airflow  AIRFLOW\_\_CORE\_\_LOAD\_EXAMPLES: "False"  volumes:  - ./dags:/opt/airflow/dags  ports:  - "8080:8080"  airflow-scheduler:  image: apache/airflow:2.6.1  restart: always  depends\_on:  - airflow-webserver  - redis  volumes:  - ./dags:/opt/airflow/dags  prometheus:  image: prom/prometheus:latest  ports:  - "9090:9090"  volumes:  - ./prometheus.yml:/etc/prometheus/prometheus.yml  grafana:  image: grafana/grafana:latest  ports:  - "3000:3000"  environment:  - GF\_SECURITY\_ADMIN\_PASSWORD=admin  statsd-exporter:  image: prom/statsd-exporter:latest  command: -statsd.mapping-config=/etc/statsd-mapping.conf  volumes:  - ./statsd-mapping.conf:/etc/statsd-mapping.conf  ports:  - "9125:9125/udp" |

Then run docker-compose up -d. Install Slack provider in Airflow:

pip install apache-airflow-providers-slack

and configure a Slack webhook via the Airflow **Connections** UI or CLI. For example, using the Airflow CLI:

|  |
| --- |
| airflow connections add slack\_conn \  --conn-type 'slack' \  --conn-host 'https://hooks.slack.com/services/T000/B000/XXXX' |

Finally, start Neo4j Desktop and create a new local graph database.

## Phase 2: Data Pipeline Foundation

* **Key deliverables:** A Python script or DAG that **extracts** metadata from Airflow (DAG definitions and run histories), **simulates** DAG executions if needed (e.g. by creating dummy DAGs or using Airflow CLI to trigger runs), and **loads** this metadata into Neo4j. This captures information like each DAG’s schedule, tasks, run timestamps, durations, and success/failure status. (We may generate synthetic data to test pipelines and anomaly detection.)
* **Tools/Libraries:** Use Airflow’s Python APIs (from airflow.models import DagBag, DagRun, TaskInstance) to programmatically access DAG structure and run records[selectfrom.dev](https://selectfrom.dev/airflow-metadata-how-to-gather-key-runtime-statistics-in-real-time-5575d8740621?gi=400f4c35458f#:~:text=Two%20critical%20Airflow%20API%20endpoints,are%20as%20follows)[selectfrom.dev](https://selectfrom.dev/airflow-metadata-how-to-gather-key-runtime-statistics-in-real-time-5575d8740621?gi=400f4c35458f#:~:text=). Use Python (and libraries like Faker or random) to create synthetic tasks/durations if needed. Use the Neo4j Python driver (neo4j package) or py2neo to write Cypher queries into Neo4j[neo4j.com](https://neo4j.com/docs/python-manual/current/#:~:text=The%20Neo4j%20Python%20driver%20is,instance%20through%20a%20Python%20application).
* **Estimated time:** ~1–2 weeks.
* **Example:**
  + Extract DAG metadata:

|  |
| --- |
| from airflow.models.dagbag import DagBag  from airflow.models.dagrun import DagRun  dagbag = DagBag('/opt/airflow/dags')  # Iterate through DAGs  for dag\_id, dag in dagbag.dags.items():  print("DAG:", dag\_id, "schedule:", dag.schedule\_interval)  # Find all runs of this DAG  dag\_runs = DagRun.find(dag\_id=dag\_id)  for run in dag\_runs:  print(run.dag\_id, run.execution\_date, run.state, run.run\_id) |

This uses Airflow’s DagBag and DagRun to pull metadata[selectfrom.dev](https://selectfrom.dev/airflow-metadata-how-to-gather-key-runtime-statistics-in-real-time-5575d8740621?gi=400f4c35458f#:~:text=Two%20critical%20Airflow%20API%20endpoints,are%20as%20follows)[selectfrom.dev](https://selectfrom.dev/airflow-metadata-how-to-gather-key-runtime-statistics-in-real-time-5575d8740621?gi=400f4c35458f#:~:text=).

* Simulate data: You might use Python to generate fake durations or task results, e.g.:

|  |
| --- |
| import random, datetime  # Generate synthetic task run durations (in seconds)  durations = [random.uniform(10, 1000) for \_ in range(100)]  for dur in durations:  # e.g., pretend these are durations for a task across runs  print("TaskDuration:", round(dur,2)) |

* Ingest into Neo4j: Use the Neo4j driver:

|  |
| --- |
| from neo4j import GraphDatabase  driver = GraphDatabase.driver("bolt://localhost:7687", auth=("neo4j", "password"))  with driver.session() as session:  # Create a DAG node  session.run("MERGE (d:Dag {id: $dag\_id, schedule: $sched})",  dag\_id=dag\_id, sched=str(dag.schedule\_interval))  # Create Task nodes and relationships  for task in dag.tasks:  session.run("""  MERGE (t:Task {id: $task\_id})  MERGE (d:Dag {id: $dag\_id})  MERGE (d)-[:HAS\_TASK]->(t)  """, dag\_id=dag\_id, task\_id=task.task\_id) |

(The Neo4j Python driver is the official way to send Cypher to Neo4j[neo4j.com](https://neo4j.com/docs/python-manual/current/#:~:text=The%20Neo4j%20Python%20driver%20is,instance%20through%20a%20Python%20application).) This code creates a :Dag node and its :Task children, linking them with :HAS\_TASK.

## Phase 3: Monitoring, Metrics & Alerting

In this phase we enable metrics and alerts. Configure Airflow to emit StatsD metrics, run a StatsD exporter, and have Prometheus scrape those metrics. For example, install Airflow with StatsD support: pip install 'apache-airflow[statsd]' and set in **airflow.cfg**:

|  |
| --- |
| [metrics]  statsd\_on = True  statsd\_host = localhost  statsd\_port = 8125  statsd\_prefix = airflow |

(as shown in the official docs[airflow.apache.org](https://airflow.apache.org/docs/apache-airflow/stable/administration-and-deployment/logging-monitoring/metrics.html#:~:text=To%20use%20StatsD%20you%20must,first%20install%20the%20required%20packages)). The statsd\_exporter container will receive Airflow’s StatsD metrics and expose them at a Prometheus endpoint[redhat.com](https://www.redhat.com/en/blog/monitoring-apache-airflow-using-prometheus#:~:text=It%20turns%20out%20that%20the,monitoring%20diagram%20looks%20as%20follows). Prometheus then scrapes this endpoint regularly (by default on port 9125), persisting the metrics. Grafana connects to Prometheus and uses dashboards to display key graphs and alerts.

* **Key deliverables:** Airflow emits metrics via StatsD; a Prometheus server is collecting those metrics via statsd\_exporter; Grafana has dashboards (e.g. DAG run times, task success rates, queue depths); Slack or email alerts are triggered on anomalies or failures. Use Alertmanager to route alerts to Slack.
* **Tools/Libraries:** Airflow’s StatsD metrics (built-in)[airflow.apache.org](https://airflow.apache.org/docs/apache-airflow/stable/administration-and-deployment/logging-monitoring/metrics.html#:~:text=To%20use%20StatsD%20you%20must,first%20install%20the%20required%20packages), Prometheus + **statsd\_exporter**, Grafana. For alerts, use Prometheus Alertmanager with a Slack webhook configured[grafana.com](https://grafana.com/blog/2020/02/25/step-by-step-guide-to-setting-up-prometheus-alertmanager-with-slack-pagerduty-and-gmail/#:~:text=global%3A%20resolve_timeout%3A%201m%20slack_api_url%3A%20%27https%3A%2F%2Fhooks). (In Alertmanager’s YAML, set global.slack\_api\_url to your webhook and a slack\_configs section with your channel[grafana.com](https://grafana.com/blog/2020/02/25/step-by-step-guide-to-setting-up-prometheus-alertmanager-with-slack-pagerduty-and-gmail/#:~:text=global%3A%20resolve_timeout%3A%201m%20slack_api_url%3A%20%27https%3A%2F%2Fhooks).) Alternatively, use Airflow’s SlackNotifier: e.g., set on\_failure\_callback=send\_slack\_notification(...) in tasks[airflow.apache.org](https://airflow.apache.org/docs/apache-airflow-providers-slack/stable/notifications/slack_notifier_howto_guide.html#:~:text=BashOperator%28%20task_id%3D,general%22%2C%20username%3D%22Airflow%22%2C%20%29).
* **Estimated time:** ~1 week.
* **Example:**
  + StatsD exporter: Launch with Docker or Go binary. A simple prometheus.yml snippet might include:

|  |
| --- |
| scrape\_configs:  - job\_name: 'airflow\_statsd'  static\_configs:  - targets: ['statsd-exporter:9125'] |

* Alertmanager Slack config (alertmanager.yml):

|  |
| --- |
| global:  slack\_api\_url: 'https://hooks.slack.com/services/T000/B000/XXXX'  route:  receiver: 'slack-notify'  receivers:  - name: 'slack-notify'  slack\_configs:  - channel: '#alerts'  send\_resolved: true |

(This matches examples on Slack webhook setup[grafana.com](https://grafana.com/blog/2020/02/25/step-by-step-guide-to-setting-up-prometheus-alertmanager-with-slack-pagerduty-and-gmail/#:~:text=global%3A%20resolve_timeout%3A%201m%20slack_api_url%3A%20%27https%3A%2F%2Fhooks).)

* Airflow Slack callbacks: In a DAG file, one can add:

|  |
| --- |
| * + from airflow.providers.slack.notifications.slack import send\_slack\_notification   task = BashOperator(  task\_id="task1",  on\_failure\_callback=[  send\_slack\_notification(  text="Task {{ ti.task\_id }} failed",  channel="#alerts",  username="airflow-bot"  )  ],  bash\_command="exit 1"  ) |

(See [60†L114-L122] for a similar example.)

## Phase 4: AI Component (Anomaly/Failure Detection)

* **Key deliverables:** Implement a basic anomaly detection on task or DAG run metrics (e.g. durations, failure counts). This could be a standalone script or an Airflow task that periodically analyzes recent data and flags anomalies. For example, use scikit-learn’s **IsolationForest** or **DBSCAN** to detect outlier runs.
* **Tools/Libraries:** Python with scikit-learn. IsolationForest and DBSCAN are unsupervised algorithms suitable for anomaly detection[scikit-learn.org](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.IsolationForest.html#:~:text=The%20IsolationForest%20%E2%80%98isolates%E2%80%99%20observations%20by,values%20of%20the%20selected%20feature)[scikit-learn.org](https://scikit-learn.org/stable/modules/generated/sklearn.cluster.DBSCAN.html#:~:text=Perform%20DBSCAN%20clustering%20from%20vector,array%20or%20distance%20matrix). Pandas/Numpy for data handling. Optionally use Airflow XCom or the Neo4j graph as data source.
* **Estimated time:** 2–3 days to prototype.
* **Example:**

|  |
| --- |
| from sklearn.ensemble import IsolationForest  import numpy as np  # Example: task durations (seconds) and failure flag as feature  data = np.array([[12.0, 0], [13.5, 0], [100.0, 1], [11.0, 0]])  clf = IsolationForest(contamination=0.1, random\_state=42)  preds = clf.fit\_predict(data)  # preds == -1 indicates anomaly (e.g., the [100,1] run)  print("IsolationForest predictions:", preds) |

The above uses IsolationForest (“isolates” outliers by random partitions[scikit-learn.org](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.IsolationForest.html#:~:text=The%20IsolationForest%20%E2%80%98isolates%E2%80%99%20observations%20by,values%20of%20the%20selected%20feature)[scikit-learn.org](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.IsolationForest.html#:~:text=Random%20partitioning%20produces%20noticeably%20shorter,highly%20likely%20to%20be%20anomalies)). Alternatively, DBSCAN can flag low-density points as noise (outliers):

|  |
| --- |
| * from sklearn.cluster import DBSCAN   durations = np.array([[12],[13],[100],[11]])  db = DBSCAN(eps=5, min\_samples=2).fit(durations)  labels = db.labels\_ # -1 for outliers  print("DBSCAN labels:", labels) |

(DBSCAN finds clusters and marks distant points as noise[scikit-learn.org](https://scikit-learn.org/stable/modules/generated/sklearn.cluster.DBSCAN.html#:~:text=Perform%20DBSCAN%20clustering%20from%20vector,array%20or%20distance%20matrix).) Any detected anomalies can trigger a Slack alert or DAG pause.

## Phase 5: Knowledge Graph Design & Ingestion

Design a Neo4j graph schema that mirrors the Airflow pipelines. For example, represent each DAG as a :Dag node, each Task as a :Task node, and use relationships like (:Dag)-[:HAS\_TASK]->(:Task) and (:Task)-[:DEPENDS\_ON]->(:Task) for task dependencies. The property graph “naturally mirrors real-world relationships”[neo4j.com](https://neo4j.com/use-cases/knowledge-graph/#:~:text=What%20is%20a%20Knowledge%20Graph%3F): DAGs, Tasks, and perhaps ExternalSystems as nodes, with properties for names, schedules, statuses, timestamps, etc. Load the DAG structure from code (as in Phase 2) and create relationships in Neo4j accordingly. This makes it easy to query data lineage or find upstream/downstream tasks.

* **Key deliverables:** A documented graph model (e.g. :Dag, :Task, and relevant relationships), and scripts to ingest data into this schema. Enable Cypher queries for insights (e.g. “find all downstream tasks of X”).
* **Tools/Libraries:** Neo4j Desktop (for UI/development) or Neo4j Aura, the Neo4j Python driver[neo4j.com](https://neo4j.com/docs/python-manual/current/#:~:text=The%20Neo4j%20Python%20driver%20is,instance%20through%20a%20Python%20application) or Cypher. (Neo4j Desktop lets you “create any number of local databases” for dev[neo4j.com](https://neo4j.com/docs/desktop-manual/current/#:~:text=Neo4j%20Desktop%20is%20a%20client,the%20resources%20of%20your%20machine).)
* **Estimated time:** ~1 week.
* **Example:**

CREATE (d:Dag {id: 'example\_dag', schedule: '\*/5 \* \* \* \*'})

CREATE (t1:Task {id: 'task\_a'}), (t2:Task {id: 'task\_b'})

MERGE (d)-[:HAS\_TASK]->(t1)

MERGE (d)-[:HAS\_TASK]->(t2)

MERGE (t1)-[:DEPENDS\_ON]->(t2);

Or in Python:

|  |
| --- |
| with driver.session() as session:  session.run("MERGE (d:Dag {id:$dag})", dag="example\_dag")  session.run("MERGE (t:Task {id:$task})", task="task\_a")  session.run("""  MATCH (d:Dag {id:$dag}), (t:Task {id:$task})  MERGE (d)-[:HAS\_TASK]->(t)  """, dag="example\_dag", task="task\_a") |

Once data is loaded, you can query the graph. For example:

|  |
| --- |
| * MATCH (t:Task)-[:DEPENDS\_ON]->(next:Task)   RETURN t.id AS task, collect(next.id) AS downstream |

This yields each task and its downstream tasks. Building this **knowledge graph** unlocks querying “interrelated data entities”[neo4j.com](https://neo4j.com/use-cases/knowledge-graph/#:~:text=What%20is%20a%20Knowledge%20Graph%3F) of the DAGs.

## Phase 6: UI Dashboard

* **Key deliverables:** A simple web dashboard that shows current DAG statuses, recent runs, and highlights critical workflows or anomalies. This could be a Streamlit app or a lightweight frontend. The goal is a quick overview: e.g. list of DAGs with last run time and state, charts of task durations, or a visual graph of dependencies.
* **Tools/Libraries:** Streamlit (for rapid Python-based UIs) or Flask/React if preferred. Streamlit “makes it easy to create web-based visualizations”[dev.to](https://dev.to/olgabraginskaya/data-analytics-dashboard-starter-kit-with-neon-streamlit-and-airflow-3onp#:~:text=The%20Data%20Analytics%20Dashboard%20Starter,used%20Airflow%20managed%20through%20Astronomer) and integrate with Python backends. Other options: a basic Node.js or HTML/JS app. Use Plotly or matplotlib for charts, or Cytoscape/D3 for graph views.
* **Estimated time:** 2–4 days.
* **Example:** A minimal Streamlit app:

|  |
| --- |
| * import streamlit as st   import pandas as pd  st.title("Airflow DAG Monitor")  # Example static data  df = pd.DataFrame([  {"dag": "example\_dag", "last\_run": "2025-06-01 10:00", "state": "success"},  {"dag": "data\_pipeline", "last\_run": "2025-06-01 09:55", "state": "failed"}  ])  st.table(df) # display status table  # Highlight failures  fail\_count = (df['state'] == 'failed').sum()  if fail\_count > 0:  st.warning(f"{fail\_count} DAG(s) have failed recently!") |

Running streamlit run app.py would show a simple table of DAG statuses and a warning if any failed. Streamlit’s simplicity is ideal for rapid prototyping[dev.to](https://dev.to/olgabraginskaya/data-analytics-dashboard-starter-kit-with-neon-streamlit-and-airflow-3onp#:~:text=The%20Data%20Analytics%20Dashboard%20Starter,used%20Airflow%20managed%20through%20Astronomer). For dependency visualization, one could use st.graphviz\_chart or a network library (e.g. PyVis) to show task graphs pulled from Neo4j. The UI ties together data from Airflow and the knowledge graph to give operators insight into the system state.

## Phase 7: Optional Enhancements

* **Auto-pause DAGs:** Implement logic to pause or throttle DAGs when resource usage is high or failures repeat. For example, write an Airflow task (or external script) that checks recent failure counts or Kubernetes CPU usage, and if thresholds are exceeded, calls airflow dags pause <dag\_id>. (Airflow’s CLI supports this, or use the REST API.) One approach is a **BranchPythonOperator** that, upon detecting N consecutive failures, triggers a BashOperator to run airflow dags pause my\_dag[stackoverflow.com](https://stackoverflow.com/questions/55255707/airflow-disable-dag-after-x-consecutive-fails#:~:text=Currently%2C%20there%20is%20no%20such,cli).
* **Policy/Rules Engine:** Add a simple rule system for automated actions. For instance, use the **Durable Rules** library, which is “a powerful, open-source rule engine” for real-time event processing[nected.ai](https://www.nected.ai/us/blog-us/python-rule-engines-automate-and-enforce-with-python#:~:text=3). You could define rules like “If DAG X has two failures in a row, then pause it”. A small example with durable\_rules:

|  |
| --- |
| * from durable.lang import \*   with ruleset('dag\_policies'):  @when\_all(m.event == 'dag\_run' & m.state == 'failed')  def on\_dag\_fail(c):  dag\_id = c.m.dag\_id  print(f"DAG {dag\_id} failed. Applying policy...")  # e.g. call Airflow API to pause dag\_id  # Post an event to the rule engine  post('dag\_policies', {'event': 'dag\_run', 'dag\_id': 'example\_dag', 'state': 'failed'}) |

(This mirrors examples on Durable Rules usage[pypi.org](https://pypi.org/project/durable-rules/#:~:text=To%20install%20the%20framework%20do%3A,pip%20install%20durable_rules).) Durable Rules can run inside a service or task to trigger actions (pause, scale up resources, etc.) based on declarative conditions.

* **Estimated time:** 3–5 days for prototypes.
* **Example:** The above snippet shows a rule reacting to a failed DAG. Using Durable Rules requires installing durable\_rules (via pip install durable\_rules) and then rules definitions like those shown[pypi.org](https://pypi.org/project/durable-rules/#:~:text=To%20install%20the%20framework%20do%3A,pip%20install%20durable_rules). This adds a lightweight policy engine to ORC AI.