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).
- **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 connectionairflow.apache.org), 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 recordsselectfrom.devselectfrom.dev. 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 Neo4jneo4j.com.
- **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 metadataselectfrom.devselectfrom.dev.

- *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 Neo4jneo4j.com.) 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 docsairflow.apache.org). The `statsd_exporter` container will receive Airflow's StatsD metrics and expose them at a Prometheus endpointredhat.com. 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, Prometheus + **statsd_exporter**, Grafana. For alerts, use Prometheus Alertmanager with a Slack webhook configuredgrafana.com. (In Alertmanager's YAML, set `global.slack_api_url` to your webhook and a `slack_configs` section with your channelgrafana.com.) Alternatively, use Airflow's SlackNotifier: e.g., set `on_failure_callback=send_slack_notification(...)` in tasksairflow.apache.org.
- **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 setupgrafana.com.)

- *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 detectionscikit-learn.orgscikit-learn.org. 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 partitionsscikit-learn.orgscikit-learn.org). 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 noisescikit-learn.org.) 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: 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 driverneo4j.com or Cypher. (Neo4j Desktop lets you "create any number of local databases" for devneo4j.com.)

- **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 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 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 prototypingdev.to. 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.
- **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 processingnected.ai. 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 usagepypi.org.) 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 shownpypi.org. This adds a lightweight policy engine to ORC AI.