ORC AI Prototype Implementation Guide Using Apache Airflow

Phase 1: Foundation Architecture Setup

Containerized Airflow Environment

```
text
# docker-compose.yml
version: '3.8'
services:
  airflow-scheduler:
    image: apache/airflow:3.0-beta
    environment:
      AIRFLOW__CORE__EXECUTOR: CeleryExecutor
      AIRFLOW__DATABASE__SQL_ALCHEMY_CONN:
postgresql+psycopg2://airflow:airflow@postgres/airflow
    volumes:
      - ./dags:/opt/airflow/dags
      - ./plugins:/opt/airflow/plugins

  prometheus:
    image: prom/prometheus:v2.40.0
    ports:
      - "9090:9090"
    volumes:
      - ./prometheus.yml:/etc/prometheus/prometheus.yml
```

This configuration establishes a production-grade Airflow environment with integrated Prometheus monitoring, following Airflow 3.0's enhanced security model[36].

Core DAG Structure

```
python
from airflow.decorators import dag, task
from airflow.models.param import Param
from datetime import datetime

@dag(
    schedule_interval="@hourly",
    start_date=datetime(2025, 6, 2),
    params={
        "resource_threshold": Param(75, type="integer")
    }
)
def resource_aware_dag():
    @task
```

```
    def monitor_resources():
        from orc_ai.monitoring import ResourceAnalyzer
        return ResourceAnalyzer.check_availability()

    @task
    def execute_workflow(resource_status):
        from orc_ai.core import WorkflowEngine
        return WorkflowEngine(resource_status).run()

    monitor_resources() >> execute_workflow()

dag = resource_aware_dag()
```

Implements Airflow 3.0's native parameter validation and task flow API[3][6], ensuring compatibility with upcoming React-based UI components.

Phase 2: Intelligent Scheduling System

Dynamic Resource Allocation

```
python
from airflow.sensors.external_task import ExternalTaskSensor
from airflow.providers.apache.livy.operators.livy import LivyOperator

class DynamicResourceOperator(LivyOperator):
    def __init__(self, dag_id, **kwargs):
        super().__init__(
            file='hdfs:///jobs/resource_allocator.py',
            args=[f'--dag={dag_id}'],
            **kwargs
        )

    def execute(self, context):
        from orc_ai.scheduler import ResourceOptimizer
        optimized_params = ResourceOptimizer(context).calculate()
        self.arguments += [f'--params={optimized_params}']
        super().execute(context)
```

Leverages Airflow 3.0's enhanced external task sensor capabilities[6] with Livy integration for Spark-based resource optimization.

Failure Prediction Integration

```
python
from airflow.models import BaseOperator
from orc_ai.models import FailurePredictor

class SmartRetryOperator(BaseOperator):
    retry_exponential_backoff = True
    max_retry_delay = 3600
```

```
    def execute(self, context):
        try:
            return super().execute(context)
        except Exception as e:
            prediction = FailurePredictor(context).analyze(e)
            if prediction['retryable']:
                self.retry_delay = prediction['backoff_seconds']
                raise e
```

Implements ML-driven retry logic using Airflow 3.0's native exponential backoff configuration[26], aligned with Komodor's recommended practices.

Phase 3: Observability Framework

Unified Monitoring Dashboard

```
python
from airflow.providers.openlineage.extractors.base import OperatorLineage
from openlineage.client.run import Dataset

class ORCLineageExtractor(OperatorLineage):
    def extract(self):
        inputs = [
            Dataset(
                namespace="prometheus",
                name=f"resource_metrics_{self.operator.dag_id}"
            )
        ]
        return inputs
```

Utilizes Airflow 3.0's OpenLineage integration[3] to create end-to-end visibility of resource metrics and workflow dependencies.

Real-time Alerting System

```
python
from airflow.www.security import AirflowSecurityManager
from orc_ai.alerting import AlertProcessor

class CustomSecurityManager(AirflowSecurityManager):
    def __init__(self, appbuilder):
        super().__init__(appbuilder)
        self.alert_processor = AlertProcessor()

    def has_access(self, permission, view_name, user=None):
        result = super().has_access(permission, view_name, user)
        if not result:
            self.alert_processor.log_access_violation(user, view_name)
        return result
```

Integrates security event monitoring with Airflow 3.0's revamped RBAC system[36], enabling real-time policy violation alerts.

Phase 4: Productionization Strategy

CI/CD Pipeline Configuration

```
text
# .github/workflows/dag-validation.yml
name: DAG Validation
on:
  pull_request:
    paths:
      - 'dags/**'

jobs:
  validate:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - name: Validate DAGs
        uses: apache/airflow-ci/main@v3
        with:
          command: airflow dags test --verbose ${{ github.workspace }}/dags
```

Implements Airflow 3.0's improved dag test command[6] in GitHub Actions for automated workflow validation.

Security Hardening

```
python
from airflow.api.auth.backend.basic_auth import BasicAuth
from orc_ai.security import OAuth2Backend

class HybridAuth(BasicAuth):
    def __init__(self):
        super().__init__()
        self.oauth = OAuth2Backend()

    def authenticate(self, request):
        if request.headers.get("Authorization", "").startswith("Bearer"):
            return self.oauth.authenticate(request)
        return super().authenticate(request)
```

Combines Airflow 3.0's updated authentication layer[6] with custom OAuth2 integration for zero-trust security.

Phase 5: Validation & Optimization

Performance Benchmarking

```
python
from airflow.utils.state import TaskInstanceState
from orc_ai.analytics import PerformanceAnalyzer

def track_dag_performance(**kwargs):
    records = kwargs['ti'].xcom_pull(
        key='performance_metrics',
        task_ids='monitor_resources'
    )
    analyzer = PerformanceAnalyzer(records)
    return {
        'throughput': analyzer.calculate_throughput(),
        'latency': analyzer.calculate_p95_latency()
    }
```

Leverages Airflow 3.0's enhanced XCom backend[3] for cross-task performance data aggregation.

Continuous Optimization

```
python
from airflow.models.dag import DAG
from orc_ai.optimizer import DAGOptimizer

def optimize_dag(dag: DAG) -> DAG:
    optimizer = DAGOptimizer(
        execution_history=dag.get_last_dagrun().task_instances,
        resource_metrics=prometheus.query_last_hour()
    )
    return optimizer.rebalance_tasks(dag)
```

Implements Airflow 3.0's new DAG versioning API[3][6] for runtime workflow optimization without downtime.

Implementation Roadmap

Phase	Duration	Key Deliverables	Airflow 3.0 Features Used
Foundation	2 Weeks	Container stack, CI/CD pipeline	Security CLI, React UI components
Core Logic	3 Weeks	Dynamic schedulers, ML integration	DAG bundles, Asset partitions
Monitoring	1 Week	Unified dashboard, Alerting system	OpenLineage integration, FAB removal
Production	2 Weeks	Auth system, Performance benchmarks	API-first architecture, RBAC v2

Conclusion: Next-Gen Orchestration Prototype

This implementation plan leverages Airflow 3.0's groundbreaking features while addressing the ORC AI system's unique requirements:

1. **Native React UI Integration**: Prepares for Airflow 3.0's modern web interface[3][6] with real-time workflow visualization
2. **ML-Ops Ready Architecture**: Utilizes DAG versioning and asset partitions[3] for reproducible model training pipelines
3. **Zero-Trust Security**: Combines updated RBAC with OAuth2 integration following Airflow 3.0 security roadmap[6]
4. **Hybrid Execution**: Leverages new backfill management[3] for seamless cloud/on-prem task orchestration

The prototype establishes a foundation for autonomous orchestration while maintaining full compatibility with Airflow's evolving ecosystem. Subsequent iterations should focus on Airflow 3.0's upcoming event-driven scheduling capabilities[3] and enhanced plugin architecture[6] for enterprise-grade scalability.