Autonomous AI Job Orchestrator: A Self-Optimizing Automation Engine for Deadline-Aware Workloads

Submitted by:

Name	**University ID No.**
J. A. L. Perera	AS2022939
A. A. Lokuliyana	AS2022921

1. Table of Contents

2. Project Overview

This project proposes the design and implementation of an AI-driven job orchestration engine that can schedule, execute, and adaptively optimize complex job workflows using reinforcement learning. The system will manage jobs with deadlines, dependencies, and priorities, execute them via a multi-threaded or asynchronous worker pool, and continuously learn from performance feedback to improve scheduling decisions over time.

3. Problem Statement

Modern systems such as data pipelines, CI/CD workflows, and backend services rely heavily on schedulers and orchestrators like cron, FIFO queues, or static priority schedulers, which are typically rule-based and manually tuned. These approaches struggle when workloads are dynamic, heterogeneous, and deadline-sensitive, often leading to high latency, resource underutilization, and job starvation under varying load conditions.

The lack of adaptation to workload changes, failure patterns, and evolving performance characteristics results in brittle automation that requires human intervention to retune parameters, reschedule jobs, or handle failures. There is a need for a self-optimizing job orchestrator that can autonomously learn effective scheduling strategies, handle failures, and maintain service-level objectives such as minimizing deadline misses and average completion time. This project addresses this problem by integrating reinforcement learning into the core scheduling loop of a distributed job orchestration engine.

4. Objectives

- Design and implement a job orchestration engine that can accept jobs with deadlines, dependencies (DAGs), and priorities, and execute them using a pool of worker threads or async workers.
- Develop and integrate a reinforcement learning–based scheduler (e.g., DQN or PPO) that selects job dispatching actions based on system state to optimize metrics such as latency, throughput, and deadline adherence.

- Implement self-healing capabilities including automatic retries, rescheduling, and rerouting of failed tasks, with rollback support for multi-step workflows.
- Build an online performance feedback loop to collect metrics and continuously update the scheduling policy during operation.
- Experimentally compare the AI-based scheduler with static baselines (FIFO, priority, and simple heuristic schedulers) in terms of latency, throughput, starvation occurrence, and learning improvement over time.

5. Scope

Scope of the project will be as following,

- Single logical orchestrator service managing a set of workers (thread pool / asyncio), with potential extension to a small, distributed setting using Ray or Celery.
- Support for DAG-style task graphs, where tasks may have dependencies and conditional execution paths.
- Reinforcement learning agent trained in a simulated or controlled environment using PyTorch, with online fine-tuning based on real execution traces.
- Basic web or CLI interface for submitting jobs, inspecting job status, and visualizing key performance metrics.

And below will be out of scope:

- Enterprise-grade security, multi-tenant isolation, or complex access control.
- Large-scale production deployment over many nodes in heterogeneous clusters.
- Very specialized domain-specific optimizations (e.g., GPU-aware placement, cost-aware multi-cloud scheduling).

6. Proposed Methodology

6. 1. Requirements Analysis (by 5 February)

- Identify typical job types, attributes (deadline, priority, estimated duration), and failure modes to be modeled in the system.
- Define functional requirements: job submission API, DAG definition model, scheduler behavior, monitoring dashboard, and failure-handling workflows.
- Define non-functional requirements: scalability targets (e.g., number of concurrent jobs), acceptable latency bounds, and reliability expectations, aligned with the course's expectations for core features and quality.

6. 2. System Design (by 20 February)

- Design an overall system architecture including:
- Orchestrator core (scheduler + state manager).
- Worker pool (threading / asyncio workers or Ray/Celery workers).
- State store (Redis for queues, job metadata, and shared state).
- RL agent module (PyTorch-based DQN/PPO) with a clear interface to the scheduler.
- Define the task graph (DAG) representation and dependency resolution mechanism.
- Design simple UI/UX artifacts (e.g., a minimal dashboard) if required, and data models for job definitions and metrics collection, as per the system design deliverables.

6. 3. Implementation Plan (February–March)

- Implement the core job orchestration engine with:
- Job ingestion API and persistence in Redis.
- DAG resolution and readiness detection for runnable tasks.
- Thread pool / asyncio workers for task execution with timeout and retry logic.
- Implement baseline schedulers (FIFO, priority, and simple heuristic) to serve as comparison points.
- Implement the RL scheduler:

- Define state features (queue lengths, deadlines, remaining work, worker utilization, recent failures).
- Define action space (which job to schedule next, or which queue to prioritize).
- Define reward function focusing on reducing deadline misses, lowering average completion time, and penalizing starvation.
- Train an initial policy offline in a simulated environment, then enable online learning using real metrics.
- Integrate self-healing mechanisms: automatic retry, backoff, fallback scheduling strategies, and simple rollback hooks for multi-step workflows.

6. 4. Testing, Validation, and Evaluation (late March)

- Develop a test plan covering unit tests (job lifecycle, failure handling), integration tests (DAG execution, scheduler–worker interaction), and stress tests under varying workload patterns.
- Design experiments to compare:
- RL scheduler vs FIFO vs static priority scheduler on:
- Average job completion time.
- Deadline miss rate.
- Throughput (jobs completed per unit time).
- Starvation incidence (jobs excessively delayed).
- Learning curve of the RL scheduler over time, showing improvement in key metrics as more experience is collected.
- Collect results, analyze trade-offs, and document limitations (e.g., exploration vs stability, overhead of learning, sensitivity to reward design).

7. Expected Outcomes and Contributions

- A working prototype of an autonomous job orchestrator that can:
- Accept and manage complex job workflows with deadlines, priorities, and dependencies.
- Schedule tasks using either conventional algorithms or an RL-based scheduler that improves with experience.

- Recover from common failures via automatic retries, rescheduling, and basic rollback mechanisms.
- Quantitative evaluation demonstrating where the AI-based scheduler outperforms static schedulers in terms of latency, throughput, and starvation reduction, as well as an analysis of its learning behavior over time.
- A structured GitHub repository with regular, meaningful commits documenting the evolution of requirements, design, implementation, and testing, aligned with the assessment plan’s emphasis on repository activity and documentation