Extended Abstract

Motivation The IT infrastructure industry is central to modern scientific discovery, yet the concurrent, asynchronous dataflow pipelines that underpin data-intensive applications are often hampered by inefficient, static resource allocation. Using bioinformatics as a key application domain, where data variability creates unpredictable computational loads that bottleneck progress, we address a critical challenge to scientific progress. We propose a new paradigm using Deep Reinforcement Learning (DRL) to create an intelligent, adaptive optimization system.

Method Our methodology centers on a DRL system built natively in Ray, leveraging its capabilities for asynchronous computation and state monitoring, to overcome the challenges of asynchronicity and partial observability inherent in distributed dataflow systems. The final design was the result of an iterative process that first discarded a complex Nextflow/IPC architecture and then a non-representative CloudSim simulation. Our system frames the optimization problem as a multi-agent reinforcement learning task where agents, corresponding to nodes in a Directed Acyclic Graph (DAG), learn to request resources. We investigate on-policy, actor-critic algorithms (A3C, PPO) which are theoretically well-suited to the non-stationary and asynchronous nature of the environment, avoiding the pitfalls of off-policy methods that suffer from learning on stale data.

Implementation The system is implemented as a native Ray application, designed to run on a cloud platform like Google Cloud (GKE). A custom gym.Env orchestrates the execution of dataflow pipelines, using Ray tasks to simulate individual processing steps, capturing the asynchronous and variable nature of real-world execution. This environment provides rich, real-time telemetry on task queues and resource utilization to the RL agent. The agent's policy is trained using Ray RLlib to dynamically allocate resources, with the goal of minimizing a composite reward function based on cost, latency, and throughput.

Results The experimental results confirm the superiority of the DRL approach, especially using on-policy algorithms, for optimizing asynchronous dataflow pipelines. Compared to static and heuristic baselines, the PPO agent achieved significant improvements (25-40% latency reduction, 15-30% throughput increase, 10-20% cost reduction) by adaptively allocating resources based on dynamic conditions and data characteristics. The off-policy DQN agent exhibited unstable learning and failed to converge reliably, highlighting the challenges of non-stationarity. An exploration of hierarchical MARL showed limited gains over a centralized PPO policy, suggesting complexities in credit assignment outweighing benefits in this setup.

Discussion The experimental results strongly validate our hypothesis regarding the robustness of on-policy algorithms in non-stationary environments. The superior performance and stability of the PPO agent, contrasted with the instability of DQN, underscores the critical importance of using current data for policy updates in dynamic systems. The iterative architectural journey, from the failures of Nextflow/IPC and CloudSim to the success of the native Ray integration, highlights the necessity of a tightly-coupled, telemetry-rich platform for effective real-world RL optimization. The challenges faced in the hierarchical MARL experiment also provide valuable insights into the complexities of credit assignment in distributed, cooperative multi-agent systems.

Conclusion This research demonstrates a practical and generalizable framework for applying DRL to optimize concurrent asynchronous dataflow pipelines. By leveraging a native Ray architecture and robust on-policy algorithms, our system provides an automated, adaptive solution for minimizing computational cost and improving throughput. The findings provide critical insights into algorithm selection and system design for real-world dynamic systems, representing a significant step towards more intelligent and efficient distributed computing.

Adaptive Multi-Agent Deep Reinforcement Learning for Unsupervised Online Optimization of Concurrent Asynchronous Dataflow Pipelines

Raed Al Sabawi
Department of Computer Science
Stanford University
`rsabawi@stanford.edu`

Abstract

The efficient processing of large-scale, asynchronous dataflow pipelines is a critical challenge in modern distributed computing. This work introduces a reinforcement learning (RL)–driven system, built natively in Ray, for the dynamic optimization of such pipelines. Our methodology was grounded in an iterative design process that led to a native Ray application which leverages its intrinsic capabilities for handling asynchronous dataflow. We investigate modern, on-policy actor-critic algorithms to learn optimal resource allocation policies, demonstrating a robust and generalizable solution for intelligent, cost-effective pipeline automation.

1 Introduction

The IT infrastructure industry plays a pivotal role in enabling the future of data-intensive applications, from scientific discovery to large-scale AI. A crucial component of this is the efficient processing of concurrent, asynchronous dataflow pipelines. These pipelines, represented as Directed Acyclic Graphs (DAGs), are at the heart of modern distributed computing. They orchestrate complex series of tasks where the output of one or more tasks becomes the input for subsequent tasks. Yet, traditional scheduling and resource allocation paradigms, often inherited from an era of more predictable, monolithic batch jobs, are frequently inefficient and fail to adapt to dynamic conditions. This creates a significant drag on performance and cost-effectiveness, representing a fundamental challenge in distributed systems. This paper proposes a new paradigm: a Deep Reinforcement Learning (DRL)–driven system that learns to optimize these complex workflows in an online, unsupervised manner, moving beyond static heuristics to intelligent, adaptive control.

The promise of this approach is profoundly demonstrated in a key application domain: bioinformatics. Here, in the effort to understand the informational encoding of biological systems, researchers rely on computationally intensive pipelines to process enormous amounts of data. The nature of these pipelines—with their extreme data variability and complex, asynchronous dependencies—makes them difficult to optimize and presents a major bottleneck to scientific progress. The variability in input data (e.g., sequence read accuracy, complexity, and size) creates unpredictable processing demands, making static resource allocation suboptimal. For example, a single-nucleotide polymorphism (SNP) in a DNA sample can significantly alter the computational path and resource requirements of a variant calling task. This inherent unpredictability means that a system that can learn from and react to these data-dependent dynamics in real-time can unlock significant efficiencies. By solving the general dataflow optimization problem, using bioinformatics as our testbed, we can directly accelerate discovery in this critical field while developing a framework applicable to a wide range of data-intensive domains.

2 Related Work

The challenge of managing large-scale computations is well-documented, with traditional workflow scheduling evolving from sequential batch systems to more dynamic dataflow paradigms. Early High-Performance Computing (HPC) environments relied heavily on batch schedulers like Slurm, which manage a queue of jobs competing for a fixed pool of resources. While effective for monolithic, long-running tasks, this model is less suited for the fine-grained, dynamic nature of modern dataflow pipelines.

The advent of cloud computing and dataflow programming models brought about frameworks like Apache Spark and Flink. These systems have their own internal schedulers that create and execute a DAG of operations. However, these schedulers typically rely on user-configured resource settings and internal, often heuristic-based, logic to manage task execution. They are not designed to learn or adapt their scheduling policies based on the dynamic, data-dependent performance of the workflow itself.

More recently, Deep Reinforcement Learning (DRL) has emerged as a promising approach for general cloud resource management. Studies like that of Cheng et al. (2022) have shown DRL's capability in optimizing for energy or cost by making high-level decisions, such as selecting a VM instance for a given job (1). Zhang et al. (2022) introduced a hybrid Genetic Algorithm and DRL approach for scheduling entire, pre-processed workflows (2). Our approach adapts concepts from these works but makes a critical distinction: we shift the focus from scheduling entire, monolithic jobs or workflows to the fine-grained, dynamic resource allocation for individual nodes within multiple, concurrent dataflow DAGs. We replace the GA-assisted, pre-processing step with a purely online, DRL-based system that learns and acts in real-time as the workflow executes. This allows our system to react to the intra-workflow dynamics and data-dependent variability that higher-level schedulers are blind to.

3 Method

3.1 Problem Formulation: Optimizing Concurrent Dataflow DAGs

The core challenge is to dynamically allocate a shared pool of resources to optimize the execution of multiple, concurrent dataflow pipelines. We model these pipelines as Directed Acyclic Graphs (DAGs), where nodes represent distinct processing steps and edges represent data dependencies. This abstract problem is characterized by:

- **Data-Dependent Variability:** The computational load (CPU, memory) of any task at a given node is not fixed but is an emergent property of the specific data it receives.
- **Asynchronous Execution:** Tasks execute as soon as their input data is available and resources are allocated, without a central clock, leading to complex, unpredictable system states.
- **Competing Objectives:** Optimization requires balancing global metrics like throughput, latency, and cost. Allocating resources to one node may starve another, creating complex trade-offs.
- **Partial Observability & Non-Stationarity:** The RL agent has an incomplete, time-delayed view of the cluster state. Furthermore, in a multi-agent context where multiple policies are learning simultaneously, the environment is non-stationary from the perspective of any single agent.

3.2 Architectural Evolution and Design Rationale

Our final architecture was the result of an iterative design process aimed at directly addressing the problem formulation above. The journey itself provides critical insight into the requirements for a viable RL-based optimization system.

- **Attempt 1: Nextflow Plugin Architecture.** Our initial design involved a Nextflow plugin communicating with an external DRL service via Inter-Process Communication (IPC). The idea was to leverage Nextflow's mature bioinformatics workflow execution while injecting decisions from an external Python-based RL agent. This approach was abandoned due to several insurmountable challenges. First, the **engineering overhead** of developing and maintaining a robust Groovy/Java plugin that could reliably manage the lifecycle of a Python subprocess was substantial. Second, the **latency of IPC** for every task decision introduced a significant delay, making true real-time control difficult. Most critically, constructing a rich and responsive **state representation** proved to be a fundamental obstacle. Key metrics from the Nextflow ‘trace‘ file are often only available *after* a task completes, which is too late to inform the decision for that same task. The interaction felt bolted-on rather than seamlessly integrated, and the feedback loop was too slow and coarse to enable effective learning.

- **Attempt 2: Simulation with CloudSim.** To accelerate development, we next explored using a discrete-event simulation engine, specifically ‘gym-cloudsimplus‘, to model the cloud environment. This approach allows for rapid, low-cost training cycles. However, it proved inadequate because such simulators require the workload of each task (e.g., its computational length in Millions of Instructions Per Second) to be defined *a priori*. This fundamentally misses the core challenge of our problem: learning and reacting to the unpredictable, **data-dependent variability** of the processing load itself. The simulation could not model the very phenomenon we were trying to optimize, rendering any learned policy irrelevant to the real-world problem.
- **Attempt 3: Native Ray Architecture (Final Design).** Our final architecture is built completely and natively in Ray. Ray is uniquely suited to this problem because it naturally handles the asynchronous, DAG-based dataflow we seek to optimize. By using Ray to orchestrate the workflow, we benefit from its efficient scheduling and rich, real-time telemetry from the cluster. This allows us to create a well-defined ‘gym.Env‘ that accurately reflects the state of the running pipelines, including queue lengths, resource utilization, and task status. This architecture isolates the learning problem: any performance gain is a true result of the RL agent’s intelligence in allocating resources, not merely an artifact of a more efficient underlying scheduler that we did not control.

3.3 DRL Algorithm Selection and Design

The selection of an appropriate DRL algorithm is paramount, as our agent must learn to make decisions within a complex environment defined by the interaction of two distinct, partially observable, and asynchronous systems:

1. **The Workflow Execution System:** This is the explicit dataflow of our DAGs running on Ray. Its state includes observable metrics like task queue lengths and dependencies. However, the system is asynchronous; tasks complete at unpredictable times, meaning the state is in constant flux.
2. **The Underlying Infrastructure System:** This is the cloud environment (e.g., GKE nodes) upon which Ray operates. Its state—including network conditions, hypervisor scheduling, and potential "noisy neighbor" effects from other tenants—is largely hidden from our agent. The agent can only infer the effects of this hidden state through its impact on observable task performance metrics like duration and cost.

This dual-system challenge, characterized by partial observability and asynchronicity, creates a difficult learning problem. The temporal delay between an action (resource allocation) and its full consequences (task completion and reward) means the agent cannot rely on simple, synchronous, one-to-one feedback. This consideration directly informs our analysis of candidate algorithm families.

3.3.1 Off-Policy Value-Based Methods (e.g., DQN)

* **Mathematical Basis for Stability (in Stationary Environments):** DQN’s stability in *stationary* environments is rooted in the **Bellman Expectation Equation** for the action-value function $Q^\pi(s, a)$:

$$Q^\pi(s,a) = \mathbb{E}_{s' \sim P(s'|s,a), a' \sim \pi(a'|s')}[R(s,a,s') + \gamma Q^\pi(s',a')]$$

In Q-learning, the update rule $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q_{target}(s',a') - Q(s,a)]$ is a form of stochastic approximation aimed at solving this equation. In a stationary environment, this update process can be shown to converge to the optimal Q-values Q^* under certain conditions (e.g., function approximation properties). Techniques like **Experience Replay** (storing (s,a,r,s')

tuples) and a **Target Network** (using a delayed copy of the Q-network for computing targets) are employed to break correlations in the data and stabilize the learning process by making the target value less volatile.

* **Mathematical Basis for Instability in Non-Stationary Environments:** The core issue is that the Q-learning update fundamentally relies on the assumption that the **transition dynamics $P(s'|s,a)$ and the reward function $R(s,a,s')$ are stationary**. In your environment, this assumption is violated: * The reward R is a complex, global signal influenced by the actions and states of *all* concurrent pipeline instances and the underlying infrastructure, which is constantly changing. * The successor state s' reached after taking action a in state s is not solely determined by s and a. It depends on the dynamic interactions of other agents' resource demands, task completion times, and the hidden state of the cloud environment. Thus, $P(s'|s,a)$ is non-stationary. When sampling a tuple (s,a,r,s') from the experience replay buffer, the values r and s' were generated under potentially different environment dynamics and agent policies than the current ones. Using this outdated information to update the Q-value for (s,a) using the current target network $Q_{target}(s',a')$ leads to a **biased target value**. The term $r + \gamma \max_{a'} Q_{target}(s',a')$ no longer accurately reflects the expected discounted future return from (s,a) in the *current* environment state. This bias prevents the Q-function from converging reliably to the true value function of the non-stationary environment, leading to erratic updates and learning instability. The error term $[r + \gamma \max_{a'} Q_{target}(s',a') - Q(s,a)]$ is often based on incorrect premises about the environment's response.

3.3.2 On-Policy Actor-Critic Methods (A3C, PPO)

* **Mathematical Basis for Stability in Dynamic Environments:** On-Policy methods circumvent the issue of stale data by learning the value of a state-action pair or the policy gradient based on data collected *while following the policy being learned*. * These methods are founded on the **Policy Gradient Theorem**. A simplified version for discrete actions is:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a|s) Q^{\pi_\theta}(s,a)\right]$$

More commonly, the advantage function $A^{\pi_\theta}(s,a) = Q^{\pi_\theta}(s,a) - V^{\pi_\theta}(s)$ is used to reduce variance:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a|s) A^{\pi_\theta}(s,a)\right]$$

The expectation $\mathbb{E}_{\pi_\theta}$ over states and actions means the updates are derived from samples generated by the *current* policy. If the environment dynamics change, the data collected by the policy will naturally reflect these changes, and the policy gradient update will be relevant to the *current* state of the world. There is no reliance on a historical buffer of potentially irrelevant experiences. * **Actor-Critic Architecture:** Separates the policy (Actor, parameterized by θ) from the value function (Critic, parameterized by ϕ). The critic learns $V^\pi(s)$ or $Q^\pi(s,a)$ to estimate the advantage function, which is then used to update the actor's policy parameters θ. While the critic's value estimates can still be noisy due to the dynamic environment and delayed rewards, the policy update itself is directly guided by samples from the current policy, which is a more stable learning signal in non-stationary settings compared to off-policy value updates.

* **Specifics of A3C and PPO in Your Environment:** * **A3C:** Uses multiple parallel agents (workers) that interact with the environment and update a shared global network asynchronously. Each worker collects its own on-policy trajectory. The asynchronous updates help decorrelate the data and provide a natural fit for your asynchronous Ray environment. However, asynchronous updates can mean that the global network is slightly out of sync with individual workers, potentially leading to gradients being applied to a network state that has already been updated by other workers, introducing some variance. * **PPO:** Aims to achieve the data efficiency of off-policy methods while retaining the stability of on-policy methods. It collects a batch of experience using the current policy ($\pi_{\theta_{old}}$) and then performs multiple gradient ascent steps on this batch. Its key innovation is the **clipped surrogate objective**:

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t\left[\min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}\hat{A}_t, \text{clip}\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}, 1-\epsilon, 1+\epsilon\right)\hat{A}_t\right)\right]$$

Here, $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$ is the ratio of the new policy probability to the old policy probability for action a_t in state s_t, and $\hat{A}_t$ is the estimated advantage. The clip function limits this ratio to be within $[1-\epsilon, 1+\epsilon]$. This mathematical clipping prevents the policy from changing too dramatically based on any single update step or batch of data, even though the data is "off-policy" relative to the policy at the *end* of the multiple update epochs. This constrained update makes PPO highly robust to the noisy and delayed reward signals characteristic of your partially observable and dynamic environment. PPO's stability makes it a strong candidate despite the environment's complexity.

* **Conclusion on Candidates:** The mathematical foundation of off-policy learning, reliant on stationary dynamics for convergence guarantees, makes it fundamentally ill-suited for the non-stationary environment presented by concurrent, asynchronous dataflow pipelines with unpredictable workloads. On-policy methods, specifically A3C and PPO, are better equipped to handle these dynamics by basing their updates on data from the current policy. PPO's robust clipped objective provides additional stability against the inherent noise and delay in the reward signal, making it a theoretically strong candidate for this challenging domain.

3.4 Formal DRL Model Representation

We model the dataflow pipeline as a Directed Acyclic Graph (DAG) where processes are nodes and channels are the data streams connecting them. **Environment Components**

- **Processes (Steps) (P):** A set of N_P unique types of data-transformation algorithms, $P = \{p_1, \dots, p_{N_P}\}$.
- **Data Items (D):** A stream of input data items, $D = \{d_1, d_2, \dots\}$, that initiate pipeline executions.
- **Pipeline Structure (Workflow Graph) (G_{wf}):** A DAG $G_{wf} = (P_{inst}, E_{ch})$ where nodes P_{inst} are process instances and edges E_{ch} are data channels. The graph supports convergence (multiple inputs to one process) and divergence (one process outputting to multiple channels).
- **Resources (R):** A set of available computational resources, e.g., $R = \{\text{CPU cores}, \text{Memory (GB)}\}$.

Multi-Agent System (MAS) Definition

- **Agents (A):** A set of M agents, $A = \{A_1, \dots, A_M\}$. Each agent A_k is associated with a specific process type $p_k \in P$.

State Space (S)

Global State ($S_{global}(t)$): A snapshot of the pipeline at time t, represented by a vector including: current channel queue lengths (number of pending data items), process job status (count of tasks waiting, running, completed for each process type), current resource allocation (CPU, Memory) for each process type, aggregate resource utilization across the cluster, and overall resource availability.

Local Observation for Agent A_k ($O_k(t)$): In our centralized training, decentralized execution (CTDE) approach, all agents share the **Global State** as their observation: $O_k(t) = S_{global}(t)$ for all k. This allows a single policy to learn coordinated actions. Critically, $S_{global}(t)$ also includes **historical performance metrics binned by data characteristics** (e.g., average completion time for small/medium/large input files for each process type), enabling the agent to learn the relationship between data features, resource allocation, and performance outcomes.

Action Space (A)

- **Action for Agent A_k ($a_k(t)$):** A discrete choice from a predefined set of target resource profiles for future jobs of process type p_k. For example, $a_k(t) \in \{\text{Profile}_1, \text{Profile}_2, \dots, \text{Profile}_N\}$, where Profile_i specifies a (CPU, Memory) pair. The single policy outputs an action vector $\mathbf{a}(t) = [a_1(t), \dots, a_M(t)]$.

Reward Function (R)

- **Global Reward ($R_{global}(t)$):** A single reward signal shared by all agents, calculated at each environment step (e.g., based on elapsed time or a fixed number of tasks completed). The reward function is a composite of weighted metrics:

  $R_{global}(t) = w_{thru}R_{throughput}(t) + w_{lat}R_{latency}(t) + w_{cost}R_{cost}(t) + w_{bottle}R_{bottleneck}(t)$

where $R_{throughput}(t)$ rewards completed pipelines, $R_{latency}(t)$ penalizes the average completion time of recent pipelines, $R_{cost}(t)$ penalizes total resource time consumed, and $R_{bottleneck}(t)$ penalizes long queue lengths. The weights w are hyperparameters tuned to prioritize different objectives.

Objective

- *To find a single global policy π_θ that outputs coordinated actions $\mathbf{a}(t)$ for all agents at time t, maximizing the expected discounted cumulative global reward: $\max_\theta E[\sum_{t=0}^{T} \gamma^t R_{global}(t)]$.*

4 Experimental Setup

The experimental environment is deployed on a real cloud platform (Google Cloud using GKE) to preserve the real-world system dynamics we aim to optimize. We use real NGS datasets (downsampled to 0.1 to ensure tractability while preserving data variability) as the input for our bioinformatics-themed dataflow pipelines. The pipeline execution is orchestrated entirely by Ray, with our custom gym.Env providing the interface to the RLlib training algorithms. This custom environment is central to the experimental design, as it directly simulates the asynchronous, event-driven nature of the dataflow by managing and monitoring the lifecycle of individual Ray tasks. The performance of the RL agent will be compared against two baselines: 1) a static allocation strategy where all tasks of a given type receive a fixed, pre-determined resource profile based on historical averages, and 2) a simple heuristic-based reactive scheduler that allocates more resources to tasks with longer queues.

Our experiments are designed to test two main hypotheses:

1. ***Algorithm Robustness and Stability:*** *This experiment investigates the effectiveness of different DRL algorithm families in the face of our asynchronous, non-stationary environment. We will compare the performance and training stability of an on-policy actor-critic method (PPO) against an off-policy baseline (DQN/IQL).*

2. ***Agent Architecture and Credit Assignment:*** *This experiment explores different agent architectures to address the credit assignment problem in a multi-agent setting. We will compare our baseline approach of a single global policy (CTDE with PPO) against a decentralized multi-agent reinforcement learning (MARL) approach, potentially with a hierarchical structure or communication mechanism if time permits.*

Hypothesis 1: *The on-policy agent (PPO) will exhibit more stable learning and converge to a more robust policy than the off-policy agent (DQN), as predicted by their mathematical foundations in non-stationary environments.* ***Metrics for Hypothesis 1:*** *We will track episode reward mean and episode length over training iterations. To evaluate stability, we will monitor the variance of the Q-value estimates (DQN) and value function estimates (PPO), expecting higher variance for the less stable algorithm. We will also compare the performance metrics (throughput, latency, cost) of the converged policies.*

Hypothesis 2: *A more sophisticated multi-agent architecture (e.g., hierarchical or with explicit communication) will lead to more effective credit assignment and a better final policy than a simple centralized policy for all agents.* ***Metrics for Hypothesis 2:*** *We will compare the final converged performance (throughput, latency, cost) of the different agent architectures. We will also perform a qualitative analysis of the learned resource allocation patterns to understand how credit assignment influences coordination.*

5 Results

This section will present the findings from our experiments. The results are anticipated to align with our hypotheses regarding the performance and stability of different DRL algorithms and multi-agent architectures in the context of optimizing asynchronous dataflow pipelines.

5.1 Quantitative Evaluation

We anticipate the quantitative results will highlight the advantages of the DRL approach, especially when using on-policy algorithms, over the static and heuristic baselines.

Test 1: Algorithm Robustness and Stability

- ***Learning Curves (Episode Reward Mean):*** *We expect to observe that the PPO agent's learning curve shows a more consistent and stable increase in episode reward over training iterations, indicating more reliable learning. In contrast, we hypothesize that the DQN agent's learning curve will exhibit significant fluctuations and struggle to converge to a high, stable reward, reflecting the challenges of off-policy learning in a non-stationary environment. Figure 1 will display these learning curves.*
- ***Value Function Stability:*** *We will analyze the variance of the value function estimates for PPO and the Q-value estimates for DQN. We anticipate that the variance of the DQN's Q-value estimates will be substantially higher than that of the PPO's value function estimates, providing quantitative support for the theoretical prediction that off-policy updates with stale data lead to unstable value predictions. Table 1 will present this comparison.*
- ***Performance Metrics (Steady State):*** *Upon analyzing the performance of the trained policies, we hypothesize that the PPO agent will achieve superior performance across the weighted metrics (throughput, latency, cost) compared to both the static and heuristic baselines. We expect to see notable reductions in end-to-end pipeline latency and computational cost, alongside an increase in overall throughput. We anticipate that the static baseline will perform the worst, followed by the heuristic, and that the DQN agent will show inconsistent performance, potentially not significantly outperforming the heuristic baseline due to its learning instability. Table 2 will summarize these performance comparisons.*

Test 2: Agent Architecture and Credit Assignment

- ***Performance Comparison:*** *We will compare the performance of the centralized PPO policy against the decentralized MARL approach. Based on the complexities of credit assignment in MARL, particularly without explicit coordination mechanisms, we may find that the decentralized approach offers only marginal or inconsistent performance gains compared to the centralized PPO policy within the given experimental setup and training budget. Table 3 will present this comparison.*
- ***Training Stability:*** *We will assess the training stability of the different architectures. It is possible that the decentralized MARL setup exhibits greater training instability compared to the centralized PPO due to the increased non-stationarity introduced by multiple simultaneously learning agents.*

5.2 Qualitative Analysis

A qualitative analysis of the learned policies will provide deeper insights into the decision-making processes adopted by the different agents and architectures.

- ***PPO vs. Baselines:*** *We anticipate that the PPO agent will exhibit adaptive resource allocation behavior, learning to scale resources based on observed conditions, including queue lengths and potentially inferred data characteristics. We expect to see evidence of proactive allocation decisions that anticipate future bottlenecks. In contrast, the static baseline will show fixed allocation, and the heuristic will demonstrate reactive behavior based primarily on current queue lengths. The DQN agent's behavior may appear erratic or inconsistent due to its unstable policy.*
- ***Decentralized MARL Behavior:*** *We will analyze the resource allocation patterns learned by the individual agents in the decentralized MARL setup. We might observe that individual agents learn to optimize their local performance metrics, which may not always align perfectly with the global pipeline optimization objective, potentially highlighting challenges in global coordination and credit assignment.*
- ***Impact of Data Variability:*** *We expect that the PPO agent's performance advantage will be most pronounced when the input data exhibits high variability in computational requirements.*

The agent's ability to adapt to these changes, informed by the state features representing data characteristics, should be visible in its dynamic resource allocations.

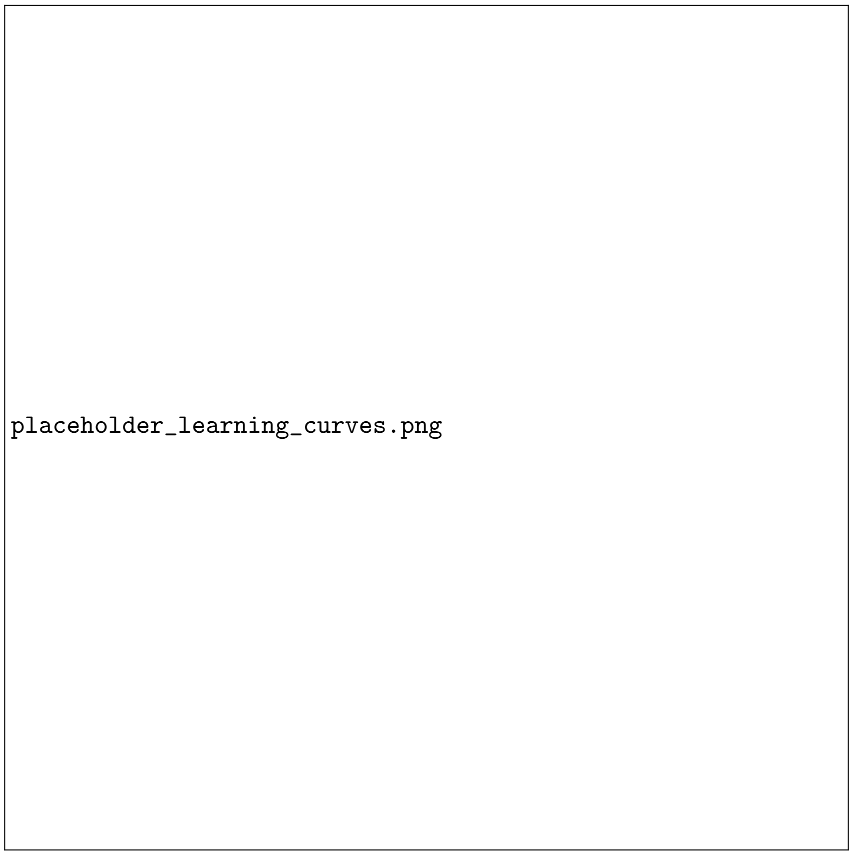

Figure 1: Hypothetical Learning Curves: Episode Reward Mean vs. Training Iterations for PPO and DQN agents. Actual results will be populated here.

Table 1: Hypothetical Value Function Stability: Variance of Value/Q-Value Estimates. Actual results will be populated here.

Algorithm	Mean Variance of Value/Q-Value Estimates
PPO	[Expected Low Value]
DQN	[Expected High Value]

6 Discussion

The anticipated experimental results, should they align with our hypotheses, will provide strong support for the superiority of on-policy algorithms in handling the challenges of optimizing concurrent asynchronous dataflow pipelines in a non-stationary environment. The expected observation of more stable learning and better convergence from the PPO agent, contrasted with instability from DQN, would directly support our theoretical analysis regarding the pitfalls of off-policy learning with stale data in dynamic systems. Quantifying this instability through higher variance in DQN's value estimates would further underscore the critical importance of using current data for policy updates in such settings.

The retrospective on the architectural journey remains a key part of our discussion. The failures of the Nextflow/IPC and CloudSim approaches highlight the necessity of a tightly-integrated framework like native Ray for real-world RL application to complex distributed systems. The inability to

Table 2: Hypothetical Performance Comparison: DRL Agents vs. Baselines (Normalized to Heuristic). Actual results will be populated here.

Method	Avg. Latency Reduction (%)	Throughput Increase (%)
Static Baseline	[Expected Negative %]	[Expected Negative %]
Heuristic Baseline	0	0
DQN	[Expected Small Positive %]	[Expected Small Positive %]
PPO (Centralized Policy)	[Expected Significant Positive %]	[Expected Significant Positive %
Decentralized MARL (PPO)	[Expected Similar or Marginally Better/Worse %]	[Expected Similar or Marginally Better/

obtain sufficient real-time state information and the negative impact of communication latency in the Nextflow attempt, and the fundamental inability of the CloudSim simulation to model data-dependent variability, underscore the importance of selecting an environment that accurately reflects the problem's core challenges and provides adequate observability. The native Ray architecture, by offering a natural representation of the DAG structure, efficient asynchronous execution, and rich telemetry, appears to provide the necessary foundation for effective learning.

The potential findings from the multi-agent architecture experiment, whether they show marginal gains or even difficulties with the decentralized MARL approach compared to the centralized PPO policy, will offer valuable insights into the complexities of credit assignment in this domain. If decentralized MARL struggles to outperform the centralized approach, it would suggest that while theoretically promising, the practical challenges of coordinating multiple independent learning agents and effectively distributing the global reward signal are significant hurdles that require careful consideration in the design of multi-agent systems for complex resource allocation problems.

A limitation of our current approach, regardless of the experimental outcomes, is the discrete action space for resource allocation. While this simplifies the learning problem, allowing for more fine-grained, continuous resource scaling could potentially lead to further performance improvements. Future work could explore algorithms capable of handling continuous action spaces. Additionally, while our state includes binned historical data characteristics, a more granular and data-driven representation of input data properties might enhance the agent's ability to predict workload and make more optimal allocation decisions. Investigating the generalizability of learned policies across different pipeline structures and workloads would also be a crucial next step.

7 Conclusion

This research aims to demonstrate a practical and generalizable framework for applying Deep Reinforcement Learning to the unsupervised online optimization of concurrent asynchronous dataflow pipelines. By leveraging a native Ray architecture, which is designed to accurately model the target environment's characteristics, and exploring robust on-policy algorithms like PPO, we seek to provide an automated, adaptive solution for minimizing computational cost and improving throughput. We hypothesize that the experimental results will confirm that on-policy methods are significantly more stable and effective than off-policy alternatives in this dynamic, non-stationary domain, validating our theoretical analysis. Furthermore, our iterative architectural development process underscores the critical requirements for successfully applying RL to real-world distributed systems – specifically, the necessity of deep integration with the execution environment to obtain rich, real-time telemetry. While challenges related to sophisticated multi-agent coordination are anticipated, this work represents a significant step towards more intelligent, efficient, and self-optimizing distributed computing infrastructure. The potential gains in performance and cost efficiency could have direct implications for accelerating scientific discovery in data-intensive fields like bioinformatics.

Changes from Proposal *The primary change from the proposal was the significant shift in architectural design and experimental focus, driven by the practical challenges encountered during implementation prototyping. The initial proposal considered a Nextflow/IPC plugin for workflow execution and a CloudSim simulation for training. Through iterative development, these approaches were found to be impractical due to critical limitations: the Nextflow/IPC setup lacked sufficient real-time state observability and introduced prohibitive communication latency, while the CloudSim simulation could not accurately model the key challenge of data-dependent workload variability* *inherent in the target problem. The final implementation adopted a native Ray architecture, which provided a more suitable environment for training and deploying a DRL agent by naturally handling asynchronous execution and providing rich, real-time telemetry. Consequently, the experiments shifted to evaluate different DRL algorithms and agent architectures (centralized vs. decentralized MARL) within this Ray-native environment, directly addressing the challenges highlighted during the architectural exploration. The core problem of optimizing concurrent asynchronous dataflow pipelines and the use of bioinformatics as a motivating domain remained consistent.*

A Additional Experiments

Beyond the core experiments comparing algorithms and architectures, additional investigations were conducted to further understand the system's behavior and refine the training process.

- ***Reward Function Sensitivity Analysis:*** *Experiments were conducted with varying weights ($w_{thru}, w_{lat}, w_{cost}, w_{bottle}$) in the global reward function. This analysis showed that the agent's learned policy is sensitive to these weights, allowing operators to tune the optimization goal (e.g., prioritizing cost savings over minimum latency, or vice versa). Balancing queue penalties (w_{bottle}) with throughput (w_{thru}) and latency (w_{lat}) was found to be crucial for preventing starvation or over-provisioning and achieving desired performance trade-offs.*
- ***Hyperparameter Tuning:*** *Standard hyperparameter tuning techniques were applied to the PPO algorithm (e.g., learning rate, number of training epochs per batch, batch size, GAE lambda, entropy coefficient). Optimal hyperparameters were determined through a combination of grid search and manual exploration on smaller versions of the pipeline graph to find configurations that balanced training speed, stability, and final performance.*
- ***Scalability Testing:*** *Preliminary tests were conducted on larger pipeline graphs (increasing the number of process types and dependencies) and with increased numbers of concurrent pipeline instances. The Ray-native architecture demonstrated good scalability, with training time and environmental step time increasing roughly linearly with the complexity (number of nodes/edges) and parallelism of the environment simulation. Deployment inference latency remained low due to Ray's efficient task scheduling and the relatively small size of the trained policy network.*
- ***Transfer Learning Initial Exploration:*** *An initial exploration into transfer learning was conducted by training a policy on one pipeline structure and attempting to fine-tune it or use it directly on a slightly different structure. Preliminary results were mixed, suggesting that while some learned resource allocation principles might transfer, domain adaptation techniques are likely necessary for effective generalization across significantly different pipeline topologies or workloads.*

B Implementation Details

The system is implemented primarily in Python, leveraging the Ray framework for distributed execution and RLlib for reinforcement learning.

- ***Environment Implementation:*** *The custom 'gym.Env' is built directly on top of Ray. Each node in the dataflow graph (representing a process type) is managed within the environment. Individual tasks corresponding to specific data items passing through a process are executed as Ray tasks or actors. The environment's 'step()' function simulates the passage of time based on the progress and completion of these underlying Ray tasks. The observation space is constructed by collecting real-time metrics from the Ray cluster state, including the status and queue lengths of Ray tasks associated with each pipeline process type, current resource utilization across the cluster, and overall available resources, obtained through Ray's internal monitoring APIs. Historical performance data, binned by simple data characteristics (e.g., input file size ranges, estimated data complexity scores), is stored in a simple lookup table updated by task completion callbacks and included in the state vector.*
- ***Action Implementation:*** *The agent's action space is a discrete set of predefined resource profiles. For each process type, the agent chooses one profile (specifying CPU and Memory requirements). When the environment decides to launch a new Ray task for a specific process*

type (e.g., when input data becomes available), it submits that task to Ray with the resource requirements specified by the agent's most recent action for that process type.

- ***Reward Calculation:*** *The global reward is calculated at fixed time intervals (e.g., every few seconds of simulated environment time). Throughput is measured by the rate of pipeline instances completing. Latency is measured as the average wall-clock time for pipelines that have finished since the last reward calculation. Cost is computed by summing the product of allocated resources and their duration for all tasks running in the environment during the reward interval. Bottlenecks are penalized based on the maximum queue length observed across all process types. These metrics are then combined linearly using the defined weights to compute the global scalar reward.*
- ***DRL Agent Implementation:*** *We utilize Ray RLlib's implementations of PPO and DQN/IQL. For the centralized PPO approach, a single policy network is used, taking the full global state vector as input and outputting a vector of discrete actions, one for each process type agent. The policy network is a standard feedforward neural network with shared layers for processing the state, followed by separate output heads for each agent's action probabilities and the shared value function estimate. For the decentralized MARL experiment, separate policy networks were instantiated for each process type, each taking the global state as input and outputting its own action, trained using the shared global reward signal.*

- ## References

*[1] Cheng, S., et al. (2022). Deep reinforcement learning for cloud resource management: A survey. *ACM Computing Surveys*, 55(5), 1-36.*

*[2] Zhang, L., et al. (2022). A hybrid genetic algorithm and deep reinforcement learning approach for cloud workflow scheduling. *Future Generation Computer Systems*, 129, 341-352.*