ENERGY-EFFICIENT CLOUD TASK SCHEDULING USING DEEP LEARNING

K. D. S. A. Chandrasiri

248054V

Master of Science (Major Component Research)

Department of Computer Science & Engineering
Faculty of Engineering

University of Moratuwa
Sri Lanka

April 2026

ENERGY-EFFICIENT CLOUD TASK SCHEDULING USING DEEP LEARNING

K. D. S. A. Chandrasiri

248054V

Thesis submitted in partial fulfillment of the requirements for the degree
Master of Science (Major Component Research)

Department of Computer Science & Engineering
Faculty of Engineering

University of Moratuwa
Sri Lanka

April 2026

DECLARATION

I declare that this is my own work and this Thesis does not incorporate without acknowledgement any material previously submitted for a Degree or Diploma in any other University or Institute of higher learning and to the best of my knowledge and belief it does not contain any material previously published or written by another person except where the acknowledgement is made in the text. I retain the right to use this content in whole or part in future works (such as articles or books).

Signature: Date: 2026-04-22

The supervisor should certify the Thesis with the following declaration.

The above candidate has carried out research for the Master of Science (Major Component Research) Thesis under my supervision. I confirm that the declaration made above by the student is true and correct.

Name of Supervisor: Prof. Dulani Meedeniya

Signature of the Supervisor: Date: 22/04/2026

DEDICATION

I dedicate this thesis to my parents, whose unconditional love, guidance, and sacrifices have been the foundation of my educational journey and personal growth. I am deeply grateful to my beloved wife for her constant encouragement, patience, and understanding, which sustained me throughout the challenges of this work.

I also extend my sincere appreciation to my supervisor, Prof. Dulani Meedeniya, for her invaluable guidance, insightful feedback, and continuous support throughout the completion of this thesis. Her mentorship has been instrumental in shaping my research and academic development.

ACKNOWLEDGEMENT

I would like to express my sincere gratitude to my supervisor, Prof. Dulani Meedeniya, for her invaluable guidance, continuous encouragement, and insightful feedback throughout the course of this research. Her expertise and support have been instrumental in shaping this work and in strengthening my academic and research skills.

I am also grateful to the academic and administrative staff of the department for their support and assistance during my postgraduate studies. Their contributions have created a conducive environment for learning and research.

Finally, I would like to thank my family and friends for their unwavering support and encouragement, which have been a source of strength and motivation throughout this journey.

ABSTRACT

This thesis presents a deep learning-based approach for energy-efficient cloud workflow scheduling by modeling scheduling as a multi-objective reinforcement learning problem that simultaneously optimizes makespan, energy consumption, and Quality of Service (QoS) . The work first introduces GNN-Flat, a baseline Graph Neural Network (GNN) scheduler that represents workflow dependencies using Directed Acyclic Graph (DAG) structures and demonstrates the feasibility of applying graph-based learning to cloud task scheduling. Building on insights from this implementation, the research proposes 2SD-GAT, a two-stage deep reinforcement learning scheduler based on Graph Attention Networks (GAT) , which forms the core contribution of the thesis. The 2SD-GAT architecture separates task selection and resource allocation decisions while leveraging attention mechanisms to capture inter-task dependencies and preference-based reward optimization, enabling improved Pareto trade-offs across competing objectives. Extensive experiments using synthetic and real-world datasets show superior performance compared with heuristic and learning-based baselines. The results show that the proposed approach achieves a 26.8% hypervolume gain with a 4.5x lower Inverted Generational Distance (IGD), along with a makespan improvement of 14.08%. Finally, the research bridges theory and practice by developing a pluggable RL-based scheduling framework, integrations with Apache Airflow and Kubernetes, and supporting tools including workflow engines, execution components, and a web-based GUI, demonstrating how the proposed scheduler can be deployed in realistic cloud orchestration environments.

Keywords: cloud, cloud-task-scheduling, deep-reinforcement-learning, energy-consumption, multi-objective optimization

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

Abbreviation	Description
ACO	Ant Colony Optimization
AI	Artificial Intelligence
CAGR	Compound Annual Growth Rate
CIS	Cloud Information Service
CLI	Command Line Interface
CP	Constraint programming
CRDs	Custom Resource Definitions
CSO	Cat Swarm Optimization
DAG	Directed Acyclic Graph
DDQN	Double Deep Q-Network
DFJSSP	Dynamic FJSSP
DRL	Deep Reinforcement Learning
DS	Deep Set
DVFS	Dynamic Voltage and Frequency Scaling
EA	Evolutionary Algorithms
EDF	Earliest Deadline First
ETL	Extract-Transform-Load
FCFS	First-Come-First-Serve
FJSSP	Flexible Job Shop Scheduling Problem
GA	Genetic Algorithms
GAT	Graph Attention Networks
GIN	Graph Isomorphism Network
GNN	Graph Neural Network
GSA	Gravitational Search Algorithm
HEFT	Heterogeneous Earliest Finish Time
HV	Hypervolume
IGD	Inverted Generational Distance
ILP	Integer Linear Programming
KinD	Kubernetes in Docker
LSTM	Long Short-Term Memory
MDP	Markov Decision Process
ML	Machine Learning
MLP	Multi-Layer Perceptron

Abbreviation	Description
MOHEFT	Multi-Objective Heterogeneous Earliest Finish Time
MOO	Multi-Objective Optimization
MTBF	Mean Time Between Failures
NIST	National Institute of Standards and Technology
PPO	Proximal Policy Optimization
PSO	Particle Swarm Optimization
QoS	Quality of Service
RL	Reinforcement Learning
RNNs	Recurrent Neural Networks
RR	Round Robin
SPEC	Standard Performance Evaluation Corporation
VM	Virtual Machine

LIST OF APPENDICES

CHAPTER 1

INTRODUCTION

1.1 Background

Cloud computing has transformed modern computing by offering scalable and on-demand access to computing resources over the internet [1, 2]. These include virtualized infrastructure, storage, and application services that cater to diverse industry needs. Among the key workloads executed in cloud environments are workflows, sets of interdependent tasks modeled as DAG [3], common in data-intensive and scientific applications [4]. Scheduling these workflows efficiently, particularly in large-scale, heterogeneous, and dynamic cloud systems, is a challenging task. Cloud workflow scheduling is inherently an NP-hard problem [5] that must balance multiple objectives, including minimizing execution time (makespan) [6], reducing energy consumption [7], and ensuring QoS.

To address these complexities, emerging solutions explore machine learning-based techniques, especially Deep Reinforcement Learning (DRL) , to dynamically adapt scheduling policies in real time [8]. Despite promising results, current approaches often fall short in supporting multi-objective optimization [9], modeling realistic workflow dependencies, and integrating with production-level workflow engines. This research aims to address these gaps through a DRL-based framework that integrates seamlessly with industrial task schedulers while optimizing for makespan, energy consumption, and QoS in dynamic cloud environments.

1.2 Motivation

Cloud datacenters are estimated to consume a growing share of global electricity, driven by rapid expansion in data-driven services and AI workloads [10, 11]. As demand scales, energy efficiency has become a crucial sustainability concern [7]. Meanwhile, the highly dynamic nature of modern cloud environments, with unpredictable task arrivals, variable resource availability, and workload fluctuations [12, 13], calls for adaptive scheduling methods that can respond in real time [14]. Traditional heuristic approaches often lack the flexibility to cope with such variability, while existing DRL-based solutions remain limited in multi-objective optimization and practical deployment [15, 16].

Moreover, industry-standard orchestration platforms like Apache Airflow and Argo Workflows [17, 18] require scheduling mechanisms that are not only effective but also pluggable and easy to integrate. A practical and intelligent scheduling tool that balances energy use and execution time can significantly enhance operational efficiency in industrial-scale cloud systems.

1.3 Research Question

This research is guided by the following key questions:

- **RQ1**: How can DRL be effectively utilized to optimize multiple objectives, specifically makespan, energy consumption, and QoS, in cloud workflow scheduling?
- **RQ2**: How can task dependency structures in workflow DAGs be effectively modeled using GNNs to enhance scheduling decisions?
- **RQ3**: How can heuristic/metaheuristic strategies be integrated with DRL to improve convergence and scheduling performance in dynamic and heterogeneous cloud environments?
- **RQ4**: How can the proposed scheduling framework be practically designed to integrate with industrial workflow orchestration platforms like Apache Airflow or Argo Workflows, ensuring real-world applicability?

1.4 Objectives

1. To design and develop a multi-objective cloud workflow scheduling algorithm that optimizes makespan, energy consumption, and QoS using DRL, combined with heuristic/metaheuristic strategies [8, 14].
2. To implement the proposed scheduling algorithm as a standalone software tool that integrates with simulation platforms (e.g., CloudSim [19]) and industrial workflow engines.
3. To evaluate the effectiveness of the proposed approach through simulations and comparative benchmarking against state-of-the-art scheduling techniques [20–22].

1.5 Problem Statement

Cloud workflow scheduling in dynamic environments remains a challenging problem due to the need to optimize multiple conflicting objectives such as makespan, energy consumption, and QoS [3, 16]. Existing DRL-based scheduling solutions often focus on single-objective optimization [23], neglect the structure of interdependent tasks in workflows, and lack integration with real-world scheduling systems. There is a critical need for a software solution that effectively addresses these limitations and bridges the gap between theoretical DRL models and practical cloud orchestration platforms.

1.6 Research Statement

This research proposes a Deep Reinforcement Learning-based software solution that integrates with industrial task schedulers to optimize cloud workflow scheduling for energy efficiency, makespan, and QoS in dynamic cloud environments.

1.7 Overview of Proposed Solution

The proposed system formulates cloud workflow scheduling as a DRL problem [8], leveraging policy-gradient-based algorithms such as Proximal Policy Optimization (PPO) [24]. The system models workflows as DAGs and captures inter-task dependencies through GNNs [25–27] to enhance decision-making. Heuristic and metaheuristic algorithms [14] are incorporated into the training process to guide exploration and improve convergence. The solution will be implemented as a modular tool with connectors to CloudSim [19] for simulation-based evaluation and APIs for integration with orchestration platforms like Airflow [17]. The performance will be evaluated on metrics such as makespan, energy consumption, and QoS, using real-world-inspired datasets [28, 29] and Pareto-based evaluation methods [30, 31].

1.8 Novelty and Contribution

- A DRL-based multi-objective scheduling framework that jointly optimizes energy, makespan, and QoS objectives in dynamic cloud environments [3, 32].
- Integration of GNNs to represent task dependencies in workflow DAGs, enhancing scheduling accuracy and adaptability [33].
- A fully modular and extensible tool designed for simulation and real-world deployment, bridging the gap between research prototypes and industrial applicability.
- Comprehensive evaluation using CloudSim and real-world traces (e.g., Google, Alibaba) with support for Pareto-based performance analysis [28, 34].

1.9 Practical Implications

The resulting solution can significantly improve resource utilization and energy efficiency in cloud datacenters [11, 35]. By supporting seamless integration with existing workflow orchestration platforms [17], it paves the way for real-world adoption in enterprise environments. The tool can serve as a foundational component in intelligent cloud resource management systems, contributing to both cost reduction and environmental

sustainability. Moreover, the system's multi-objective optimization capability allows operators to flexibly adapt scheduling strategies to changing priorities or constraints.

CHAPTER 2

LITERATURE SURVEY

Cloud computing has become a cornerstone of modern digital infrastructure, enabling on-demand access to scalable, virtualized computing resources through utility-based service models [1]. This paradigm shift has empowered organizations to deploy large-scale, data-intensive applications without the burden of maintaining physical infrastructure. Among these applications, scientific and industrial workflows stand out due to their structured, dependency-aware nature and growing relevance in domains such as astronomy, genomics, climate modeling, and financial analytics [36].

A workflow typically consists of multiple interdependent tasks modeled as a DAG, where edges represent data and control dependencies [3]. Scheduling such workflows in cloud environments is a complex task that involves mapping each task to a suitable Virtual Machine (VM) while respecting constraints such as deadlines, resource availability, budget limitations, and QoS requirements [14]. The inherent heterogeneity and dynamic behavior of cloud resources further increase the difficulty of making optimal scheduling decisions.

The workflow scheduling problem is widely recognized as NP-hard [5], implying that exact optimization becomes impractical for real-world scenarios at scale. Over the past decade, a wide spectrum of scheduling approaches has been proposed. Traditional heuristic and meta-heuristic algorithms have achieved significant progress in reducing makespan and execution cost with reasonable computational efficiency [14]. However, the rapid expansion of cloud infrastructure has intensified concerns regarding energy consumption, environmental sustainability, and operational cost, leading to increased focus on energy-aware scheduling [37].

In addition to classical optimization, recent advancements in Artificial Intelligence (AI) , particularly Machine Learning (ML) and DRL, have opened new research avenues [8]. These approaches can adapt to dynamic workload conditions and learn optimal decision-making strategies directly from execution feedback. Furthermore, the incorporation of multi-objective optimization enables schedulers to capture complex trade-offs, particularly between performance and sustainability [38].

Despite its conceptual importance, Pareto-awareness appears to receive limited treatment in much of contemporary cloud scheduling research, including within the rapidly growing body of DRL-based approaches [3, 39, 40]. One possible reason is the substantial computational burden associated with identifying and maintaining diverse sets of non-dominated solutions in high-dimensional state-action spaces [41]. From an industrial green cloud perspective, this tendency may have practical consequences. Without clear visibility of the Pareto front, decision-makers may struggle to accurately assess trade-offs between energy efficiency and other objectives such as

QoS guarantees [42]. This reduced transparency can potentially encourage either overly conservative scheduling that underutilizes sustainability opportunities, or aggressive energy optimization that risks degrading system reliability and performance [43].

2.1 Background

2.1.1 Energy Consumption in Cloud Computing

Cloud computing has emerged as a transformative force in modern computing, providing scalable, versatile resources and driving significant technological advancements across industries [44]. According to the National Institute of Standards and Technology (NIST) , cloud computing is defined as a model that enables easy and on-demand access to a shared pool of configurable computing resources, such as networks, servers, storage, applications, and services, that can be rapidly provisioned and released with minimal management or involvement from the service provider [1]. Cloud computing has fundamentally reshaped how organizations manage infrastructure, deploy applications, and scale their operations. By abstracting physical hardware and delivering resources as services over the internet, it reduces capital expenditure, accelerates innovation, and enables global accessibility [2]. This paradigm empowers businesses of all sizes to focus on their core competencies while leveraging powerful computing platforms on demand. As digital transformation becomes central to competitive strategy, cloud computing continues to serve as a critical enabler, driving agility, efficiency, and resilience in today's fast-evolving technological landscape [45].

As cloud computing continues to evolve, deployment models such as public, private, community, and hybrid clouds offer tailored solutions to meet diverse organizational needs [1]. In a public cloud, third-party providers deliver computing resources over the internet using a pay-per-use or subscription-based pricing model, enabling rapid scalability and cost efficiency. A private cloud, by contrast, dedicates hardware and software resources exclusively to a single organization, offering greater control, security, and customization. The community cloud model supports collaboration by allowing several organizations with shared concerns, such as compliance or mission objectives, to jointly utilize cloud infrastructure, whether managed internally or externally. Meanwhile, a hybrid cloud integrates public and private clouds (and often on-premise infrastructure) to form a unified, flexible computing environment. These deployment models are widely used across a spectrum of applications, from long-running background jobs and dynamic web services to data-intensive scientific workflows, offering organizations the ability to optimize for performance, cost, and compliance simultaneously.

With these various offerings, the global cloud computing market is expanding rapidly with a Compound Annual Growth Rate (CAGR) of 12.0%. Market forecasts predict that the market, valued at USD 1294.9 billion in 2025, will reach USD 2281.1

billion by 2030 [46]. However, this rapid growth comes at a cost, particularly in energy consumption within data centers, which are critical for cloud services. In 2022, data centers in the United States, European Union, and China accounted for significant electricity demand, with expectations for substantial growth across all regions by 2026 [11].

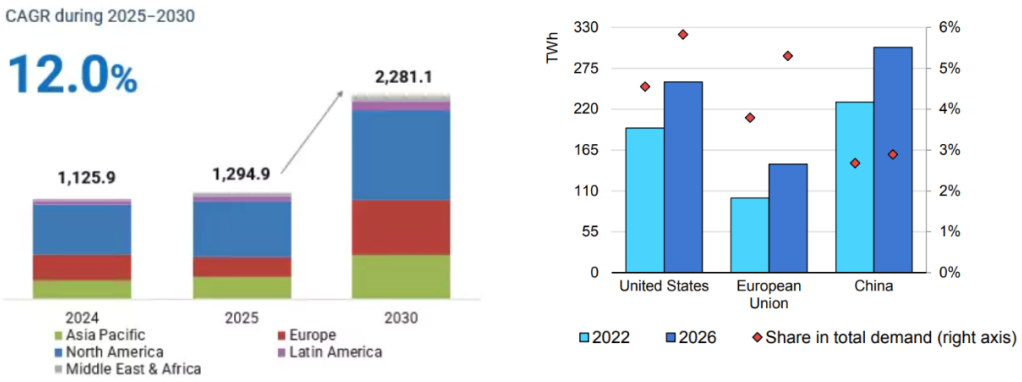

Fig. 2.1: CAGR of the Cloud Computing Market and Energy Consumption Share of Data Centers in Selected Regions

Note. Left: CAGR of the Cloud Computing Market (Taken from [46]). Right: Energy Consumption Share of Data Centers in United States, European Union, and China (Taken from [11]).

Given this trend, minimizing power consumption has become a critical concern in data centers [35]. IT equipment – including CPUs, memory, and storage devices – accounts for nearly 45% of total energy usage, making it a primary target for optimization [35]. Benchmarks from standardized evaluation tools such as SPECpower further demonstrate a strong correlation between CPU workload and energy usage, showing that power consumption rises with increasing computational demand [47].

This underscores the importance of efficient server load management, where tasks are distributed across computing resources to maintain both performance and power efficiency. As data centers continue to scale with the rapid growth of cloud-based services, optimizing CPU utilization not only for speed but also for sustainability becomes increasingly essential. Strategic scheduling and intelligent workload placement provide practical approaches to reducing energy consumption in modern cloud infrastructure [37, 48].

Beyond server-level load management, researchers have explored additional strategies to improve energy efficiency in cloud environments. Geographical workload shifting leverages the distributed nature of data centers by migrating tasks to regions with lower energy costs, reduced carbon intensity, or more favorable cooling conditions [49]. Complementary methods such as virtualization and VM consolidation [39] enable the aggregation of workloads onto fewer physical servers, allowing unused

machines to be powered down. Moreover, Dynamic Voltage and Frequency Scaling (DVFS) [22] dynamically adjusts hardware performance to match real-time demand. Advancements in cooling systems and the integration of renewable energy further contribute to reducing the environmental footprint of large-scale data centers [3].

2.1.2 Cloud Architecture and Cloud Task Scheduling

Cloud architecture leverages virtualization, where a hypervisor divides a physical host server into multiple isolated VMs. Each VM operates independently, running its own operating system and applications/tasks as if it were a standalone server [50]. Within cloud environments, VMs serve as execution units for a diverse range of application types. To effectively allocate and execute these workloads, cloud systems rely heavily on schedulers, which are responsible for mapping tasks to VMs based on available resources, execution priorities, and optimization goals such as cost, latency, or energy efficiency.

In particular, the assignment of tasks in cloud environments is handled by the scheduler with the goal of optimizing resource allocation. This challenge, referred to as the Cloud Task Scheduling Problem, is an NP-hard combinatorial optimization problem [5]. NP-hard or Non-deterministic Polynomial-time hard problems are characterized by their computational intractability — that is, no known algorithm can solve all instances of such problems in polynomial time. As the size of the input increases, the time required to compute an exact solution grows exponentially, rendering brute-force or exact approaches impractical for real-world scenarios, especially in dynamic and large-scale cloud systems.

Given this computational complexity, a wide range of approximation algorithms have been explored. These include heuristics, which use problem-specific rules to quickly generate good-enough solutions; meta-heuristics, which provide generalized search strategies for exploring large solution spaces; and machine learning-based methods, including reinforcement learning and deep learning techniques. These approaches aim to produce near-optimal solutions within acceptable timeframes [8, 14].

2.1.3 DAG-based Cloud Workflow Scheduling

The growing relevance of DAG-based scheduling is closely tied to how modern computing workloads are evolving. Today's cloud platforms increasingly support large-scale, data-intensive and dependency-aware applications [51] in domains such as astronomy, genomics, climate science, and financial analytics [36], where tasks are naturally structured with precedence and data dependencies [52]. Representing these applications as DAGs allows schedulers to explicitly model both data and control dependencies among tasks, which is essential for correctness and performance at scale [8].

At the same time, cloud environments have become more heterogeneous, dynamic, and geographically distributed, spanning core clouds, edge nodes, and hybrid infrastructures [53]. In such settings, simple independent-task models are no longer sufficient; scheduling must account for task ordering, communication, and resource variability [54]. DAG-based models provide the structural foundation to reason about these dependencies while optimizing across multiple objectives [52]. This is particularly important as sustainability and energy efficiency have emerged as first-class concerns in rapidly expanding cloud infrastructures, pushing scheduling research toward energy-aware and multi-objective formulations.

As a result, DAG-based scheduling is no longer just a theoretical construct but a necessary framework for capturing the structure, scale, and sustainability constraints of contemporary cloud workloads.

2.1.4 Deep Reinforcement Learning

DRL, a branch of machine learning that leverages deep neural networks, enables agents to learn optimal actions by interacting with their environment [55]. In DRL frameworks, an agent observes the current state, selects actions, and receives feedback in the form of rewards, which guide its learning toward long-term objectives (Figure 2.2). DRL has shown promise in addressing complex resource allocation problems, with several studies applying DRL techniques to the Cloud Task Scheduling Problem [8]. Unlike heuristic or rule-based methods, DRL agents learn from environment feedback and adapt to new scenarios without requiring hand-crafted decision rules, making them particularly suitable for highly dynamic and uncertain cloud systems. However, most existing research focuses on independent tasks, leaving the application of DRL to dynamic workflows largely underexplored.

This gap can be attributed to the unique challenges of workflow scheduling, where tasks are interdependent and must follow precedence constraints. Unlike independent task scheduling, which focuses on assigning single tasks to VMs, workflow scheduling involves multiple layers of decision-making: selecting the workflow to process, determining the next executable task within it, and assigning the task to an appropriate VM. This additional complexity introduced by workflows challenges traditional approximation methods.

2.1.5 Graph Neural Networks

Recently, GNNs have gained attention for their capability to capture dependencies in graph-structured data. While the studies integrating GNNs with DRL for workflow scheduling in cloud environments remains limited, GNNs have demonstrated success in related fields such as the Flexible Job Shop Scheduling Problem, where they effectively model complex task dependencies [33, 56, 57]. In particular, architectures

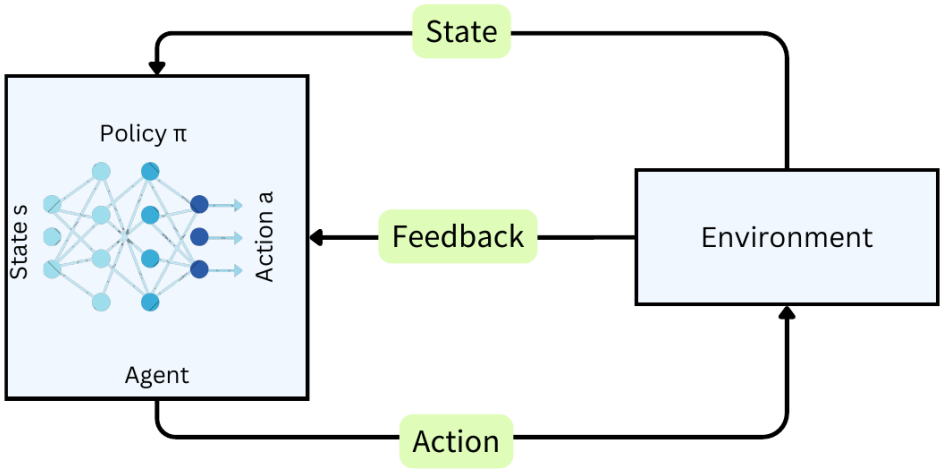

Fig. 2.2: Interaction flow of a reinforcement learning framework

Note. The agent observes the state s of the environment and selects an action a based on its policy π. The action influences the environment, resulting in feedback and a new state, enabling the agent to iteratively improve its decision-making process.

like the Graph Isomorphism Network (GIN) [26], known for its expressive power in distinguishing graph structures, and the GAT [27], which uses attention mechanisms to prioritize important neighbors, have shown promise in learning nuanced task relationships. This suggests that combining GNNs, especially advanced variants like GIN and GAT, with DRL could enhance decision-making in dynamic workflow scheduling, enabling optimized schedules that respect task dependencies and execution constraints.

2.1.6 Multi-Objective Optimization

Cloud resource management and many real-world decision-making problems often involve the simultaneous optimization of multiple, conflicting objectives. These objectives, such as performance, cost, reliability, and energy efficiency, frequently trade off against one another, meaning that improving one may lead to the deterioration of another. In such cases, it is typically not possible to identify a single globally optimal solution. Instead, Multi-Objective Optimization (MOO) offers a principled framework to evaluate and balance these trade-offs [31].

Rather than seeking a single best outcome, MOO aims to find a set of Pareto-optimal solutions. A solution is Pareto-optimal if no other solution exists that improves at least one objective without worsening another. The collection of all such non-dominated solutions forms the Pareto front, representing a diverse set of trade-off solutions among competing objectives [31]. In the context of cloud workflow scheduling, for instance, reducing makespan may require the use of high-performance resources, which increases energy consumption, highlighting the importance of considering trade-

offs when optimizing across multiple goals.

To evaluate the effectiveness of MOO algorithms, researchers use several widely accepted performance indicators. Two of the most popular are Hypervolume (HV) and Inverted Generational Distance (IGD) [30]. Hypervolume measures the size of the space covered by the set of Pareto-optimal solutions — larger hypervolume values generally indicate better coverage and diversity of trade-offs. IGD, on the other hand, assesses how close the generated solutions are to a reference set of ideal trade-offs, reflecting both accuracy and spread. Together, these indicators provide a practical way to assess how well an algorithm balances and explores conflicting objectives.

2.2 Related Cloud Scheduling Taxonomies

2.2.1 Types of Cloud Workloads

Cloud scheduling encompasses a wide spectrum of workload types, each characterized by distinct structures, execution patterns, and resource requirements [14, 58]. Common workload categories include batch workflows, bag-of-tasks workloads, long-running tasks, and managed jobs.

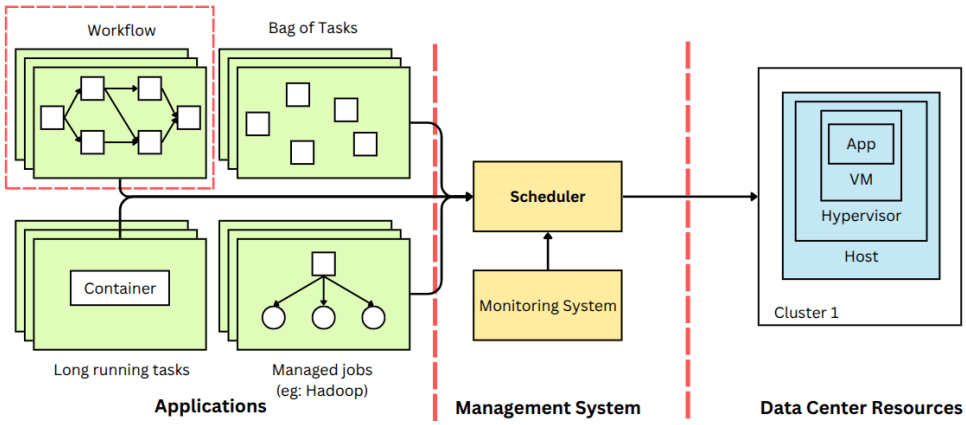

Fig. 2.3: Different types of cloud workloads and the basic structure of a cloud scheduler

Batch workflows consist of interdependent tasks typically modeled as DAGs, where the execution of each task depends on the completion of its predecessors [58]. Within this category, it is crucial to differentiate between scientific workflows and business workflows [4]. Scientific workflows are typically compute-intensive and predictable, often characterized by static structures and well-defined resource requirements suitable for offline scheduling [36]. Conversely, business workflows are predominantly data-intensive and unpredictable, exhibiting dynamic arrival patterns and bursty resource demands that complicate energy management [59]. These diverging characteristics result in significantly different energy profiles and multi-objective constraints, necessitating distinct scheduling strategies for each domain. In this research, we focus primarily

on research that aligns with scientific workflows. Scheduling batch workflows involves not only task-to-resource assignment but also the coordination of task dependencies and communication, with minimizing makespan being a common objective [14].

Bag of tasks refers to a collection of independent tasks that do not require communication or synchronization during execution. These workloads are common in applications such as Monte Carlo simulations and parameter sweeps [60]. Due to their independence, tasks in a bag of tasks can be scheduled in parallel without concern for execution order, allowing for efficient load balancing and resource utilization. Although they lack internal dependencies, they can still be modeled as trivial workflows composed of a flat set of tasks [58].

Long-running tasks are continuous or persistent computational processes that execute over extended durations. Examples include server-based applications, such as web services. **Managed jobs** refer to workloads coordinated by job management or orchestration systems such as Kubernetes, Apache Airflow, Apache Hadoop, or Apache Spark. These platforms abstract the complexities of execution, handling aspects like task scheduling, dependency resolution, retry policies, and resource provisioning.

In this research, we focus specifically on batch workflows due to their dependency-aware structure and relevance in complex, data-driven applications. This focus allows us to explore the challenges of dependency-aware scheduling, resource optimization, and energy efficiency in dynamic cloud environments.

2.2.2 Cloud Scheduling Constraints

Effective scheduling in cloud environments must address a range of system-level and application-level constraints that influence both the feasibility and efficiency of task execution. These constraints arise from the heterogeneous and dynamic nature of cloud infrastructure, as well as the structural complexity of modern cloud workloads.

Resource heterogeneity refers to the variation in computational capabilities, memory, network bandwidth, and energy profiles across different virtual machines or physical nodes within the cloud. These differences must be taken into account when assigning tasks to resources to avoid performance degradation and ensure efficient resource utilization [14].

Task dependencies introduce precedence constraints between tasks, typically in the form of DAGs. In such structures, a task cannot begin execution until all of its predecessors have completed. Scheduling in the presence of dependencies requires accurate modeling of inter-task communication and ordering, which significantly increases the complexity of workflow orchestration [6].

In many scenarios, **Resource constraints** further restrict the scheduler's flexibility. Such constraints may require certain tasks to be executed on specific instance types, based on requirements such as minimum CPU cores, memory, or specialized hardware

capabilities.

Deadline constraints are also common, especially in time-critical applications, where the entire workflow must complete before a specified deadline. Missing such deadlines may lead to quality degradation or SLA violations, making deadline-aware scheduling a key requirement in such contexts [14].

Budget constraints impose an upper limit on the total cost incurred during workflow execution. These are particularly relevant in utility-based cloud computing models, where users provision resources under cost limitations. The scheduler must therefore optimize performance while ensuring that monetary costs remain within the predefined budget [14].

2.2.3 Objectives in Cloud Workflow Scheduling

Cloud task/workflow scheduling algorithms are designed to optimize one or more performance objectives based on application requirements, user expectations, and infrastructure capabilities. These objectives often conflict, requiring trade-offs and the application of MOO techniques [14].

In cloud workflow scheduling algorithms, **makespan minimization** is a key objective that refers to the total time required to complete all tasks in a workflow. Reducing makespan improves resource utilization and system responsiveness, which is vital for high-performance and deadline-sensitive applications [6, 16].

As mentioned in the Chapter 1, **Energy consumption minimization** has become essential due to the rising operational costs and environmental concerns associated with large-scale cloud data centers. Energy-aware strategies reduce power usage through workload consolidation, energy-efficient resource selection, and machine idle state management [3, 22, 61, 62].

In commercial cloud settings, **SLA adherence** is vital, as providers guarantee various performance levels. SLAs can take many forms, including constraints on machine availability, execution deadlines, and quality parameters [16, 63, 64]. As mentioned in Section 2.2.2, violations may result in financial penalties and reduced customer satisfaction, so scheduling must ensure tasks meet the defined requirements.

The widely used objective of **minimizing monetary cost** focuses on reducing expenses associated with cloud resource usage, such as compute, storage, and data transfer. This becomes particularly important in pay-as-you-go cloud environments, where scheduling decisions can significantly impact the total cost incurred by users or providers [20, 65, 66].

Several other key objectives influence cloud workflow scheduling strategies. **Reliability maximization** aims to ensure successful task execution despite potential hardware, software, or network failures, often using techniques such as task replication, checkpointing, and failure-aware scheduling [67–69]. **Load balancing** seeks to distribute

tasks evenly across available resources to prevent bottlenecks, enhance parallelism, and avoid overloading specific nodes, thereby improving system predictability and throughput [5, 15, 70]. **Response time minimization** focuses on reducing the delay between task submission and completion, which is essential for real-time and interactive applications where low latency is critical to user experience. Meanwhile, **throughput maximization** targets an increased number of completed tasks or workflows per unit time, which is particularly relevant for high-throughput computing and big data workloads with continuous job execution.

In many real-world scenarios, scheduling algorithms must consider multiple objectives simultaneously. Multi-objective optimization frameworks are employed to achieve Pareto-optimal solutions that balance trade-offs, such as minimizing makespan while also reducing energy consumption or operational cost.

2.2.4 Static vs. Dynamic Scheduling

Cloud workflow scheduling algorithms are commonly categorized into static and dynamic approaches, based on the time at which scheduling decisions are made relative to workflow execution.

In **static scheduling**, the entire workflow is analyzed and scheduled prior to execution. All scheduling decisions, such as task-to-resource mapping and execution order, are computed offline, assuming complete knowledge of task execution times, data dependencies, and available resources [71]. Such algorithms are particularly effective in stable, predictable environments where workflow characteristics and resource availability remain unchanged during execution. Algorithms such as Min–Min, Max–Min, Heterogeneous Earliest Finish Time (HEFT) , and Multi-Objective Heterogeneous Earliest Finish Time (MOHEFT) operate under this model [6, 20]. Static methods can yield high-quality schedules with minimal makespan or cost if task runtimes and resource capabilities are accurately estimated. However, they lack adaptability to runtime dynamics such as resource contention, performance variability, or task failures.

In contrast, **dynamic scheduling** defers decision-making to runtime, allowing the scheduler to react to the current system state and workflow progress [71]. These strategies are essential in heterogeneous and volatile cloud environments where conditions frequently change. Examples include Earliest Deadline First (EDF) and Round Robin (RR) , which allocate tasks based on temporal attributes or rotational policies. More recent approaches, such as RL-based scheduling, have also gained popularity for their ability to learn optimal scheduling policies through environmental interaction. While dynamic scheduling introduces additional computational overhead, it enables better responsiveness and resilience in uncertain execution scenarios [71].

2.2.5 Preemptive vs. Non-preemptive Scheduling

Workflow scheduling strategies can also be classified based on task execution interruptibility, as either preemptive or non-preemptive. This classification determines whether a task, once started, can be paused or migrated before completion.

In **non-preemptive scheduling**, once a task begins execution on a resource, it runs to completion without interruption. This model simplifies resource management and reduces overhead [71]. Most workflow scheduling algorithms adopt a non-preemptive model due to its simplicity [6, 16, 20, 65, 72].

In contrast, **preemptive scheduling** allows a task to be paused or migrated during execution, often to accommodate higher-priority tasks, improve load balancing, or adapt to failures and dynamic changes in resource availability. This flexibility introduce overhead due to context switching, state saving, and possible data transfer. Preemption is more commonly used in operating system schedulers, real-time systems, and emerging cloud-native frameworks that support containerized or checkpointed tasks.

2.2.6 Online vs. Offline Scheduling

Workflow scheduling algorithms can also be distinguished based on the availability of task information at scheduling time. This leads to the classification into online and offline approaches.

In **offline scheduling**, all tasks and their attributes, such as execution time, dependencies, and data transfer costs, are known in advance [14]. The scheduler has full visibility of the entire workflow or a batch of workflows, which allows for global optimization. Offline algorithms typically generate a complete schedule before execution begins and are well suited for scientific workflows with predictable structures and resource requirements. Classical algorithms such as HEFT and various meta-heuristic-based methods operate under the offline model.

A common variant of offline scheduling is the batch mode [71], where incoming tasks or workflows are first collected into a queue over a defined time window and then scheduled together as a group. This approach provides a balance between scheduling efficiency and responsiveness, especially in environments where workflows arrive frequently but do not require instant scheduling decisions.

In contrast, **online scheduling** operates under incomplete information and makes scheduling decisions as tasks or workflows arrive over time. Tasks are scheduled immediately upon arrival or within a short decision window, often without knowledge of future tasks [14]. While online scheduling enables rapid responsiveness, it often results in suboptimal global performance due to limited foresight.

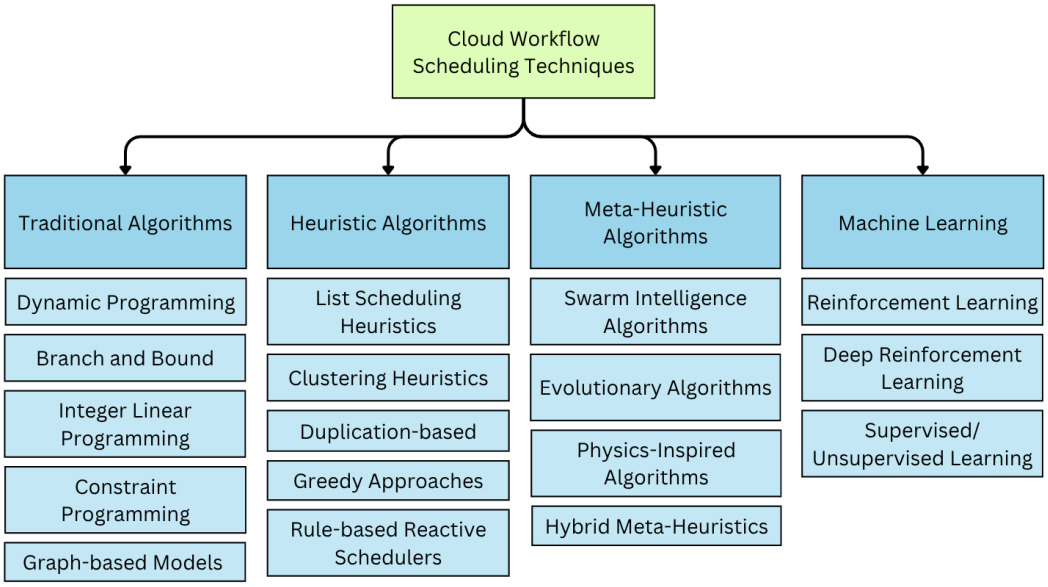

Fig. 2.4: Overview of Cloud Workflow Scheduling Algorithm Taxonomy

2.3 Energy-Efficient Cloud Workflow Scheduling Techniques

Cloud workflow scheduling algorithms can be broadly classified into four categories: traditional algorithms, heuristic methods, metaheuristic approaches, and machine learning-based techniques. This categorization reflects the algorithmic foundations and trade-offs among solution quality, computational efficiency, and adaptability to dynamic cloud environments.

Cloud workflow scheduling algorithms can be broadly classified into four categories: traditional algorithms, heuristic methods, metaheuristic approaches, and machine learning-based techniques. This categorization reflects the algorithmic foundations and trade-offs among solution quality, computational efficiency, and adaptability to dynamic cloud environments. In the following subsections, we examine each category in detail, with primary emphasis on scheduling algorithms that explicitly consider energy efficiency as one of their optimization objectives. Wherever possible, examples are drawn from energy-aware or energy-efficient scheduling studies. However, to ensure comprehensive coverage of the field and to clearly illustrate the characteristics of each category, a limited number of representative algorithms that do not directly target energy optimization are also discussed, particularly in cases where energy-aware variants are scarce or unavailable, and when there exists a notable algorithm that has significantly influenced or inspired the design of many subsequent scheduling approaches.

2.3.0.1 Traditional Approaches

Traditional algorithms are grounded in classical optimization techniques and typically assume complete knowledge of the workflow structure, task execution times, communication costs, and resource availability. These methods aim to find optimal or near-optimal schedules by formulating the problem as a mathematical model, which is then solved using deterministic procedures [14].

Dynamic programming approaches solve the scheduling problem by recursively breaking it into smaller subproblems and combining their solutions. They are well suited for workflows with structured dependencies and well-defined cost models, but suffer from the curse of dimensionality. **Branch and bound techniques** explore the search tree of possible task-to-resource assignments using upper and lower bounds to prune infeasible or suboptimal branches early. These methods can be effective for small workflow graphs, but their exponential growth limits practical applicability [14].

Integer Linear Programming (ILP) formulates the workflow scheduling problem as a set of linear equations and inequalities with integer decision variables representing task-resource assignments. Solutions to ILP models yield optimal schedules when feasible, but solving them becomes computationally intractable as the number of tasks and resources increases. **Constraint programming (CP)** models, in contrast, use logic-based constraints rather than linear equations to represent dependencies, resource capacities, and timing constraints. CP approaches are flexible and expressive, especially for multi-resource environments and temporal constraints, but their performance heavily depends on the quality of the constraint solver and problem encoding [73].

Despite their theoretical appeal, most traditional methods rely on exhaustive or near-exhaustive search strategies, making them unsuitable for large-scale workflows due to the NP-hard nature of the problem. For example, scheduling a workflow with just 20 tasks across 5 heterogeneous virtual machines can lead to billions of possible task-to-resource combinations when accounting for task dependencies, execution times, and communication costs. As the problem size grows, the search space increases exponentially, rendering exact optimization techniques computationally infeasible for real-world cloud environments with dynamic workloads and resource availability [14].

TABLE 2.1: SELECTED APPROACHES USING TRADITIONAL ALGORITHMS - NOT ENERGY-AWARE

Algorithm	**Objectives**	**Category**	**Wo**
Genez et al. [63]	Cost, Deadlines	ILP + Heuristics	Yes
MOILP [73]	Makespan, Cost	ILP	Yes
Prob [74]	Makespan	Probabilistic Modeling	Yes

Note. ILP = Integer Linear Programming, Wo = Workflow Oriented

Among traditional scheduling approaches, most formulations focus on relatively

straightforward optimization goals and are generally impractical at scale due to the NP-hard nature of workflow scheduling. Consequently, limited attention has been given to energy-aware objectives in this category. Instead, existing studies typically address other concerns such as makespan, monetary cost, and robustness. For example, [63] introduce ILP formulations with relaxed heuristics to reduce cost while meeting deadline constraints under a two-level SLA model. Likewise, MOILP [73] proposes an ILP-based model to minimize workflow makespan and cost in multi-cloud environments. Additionally, probabilistic methods, such as Prob [74], incorporate uncertainty by modeling cloud dynamics and failure behavior, aiming to enhance scheduling resilience. However, these techniques still face significant scalability challenges, limiting their practicality for large workflow deployments.

2.3.0.2 Heuristic Approaches

Heuristic algorithms rely on simplified, rule-based logic to make scheduling decisions efficiently, often without guaranteeing optimality. These methods are designed to strike a balance between scheduling quality and computational overhead, making them particularly suitable for large-scale workflows where exhaustive search or mathematical optimization becomes impractical. Heuristics typically operate using priority rules, task rankings, or cost functions derived from workflow structure and resource capabilities [71].

A wide range of heuristic strategies have been proposed to allocate workflow tasks to resources effectively with minimal computational cost. One of the most widely adopted categories is **List Scheduling Heuristics**, where tasks are prioritized using static or dynamic metrics such as upward rank or critical path length, and then assigned to resources accordingly. Another common class is **Greedy Heuristics**, which iteratively assign tasks to the most suitable resource based on local optimization goals like earliest completion time or minimum cost. Classic approaches such as Min–Min and Max–Min [71] follow this strategy by selecting task-resource pairs that either minimize completion time or better utilize slower resources.

In addition to these, several heuristics focus on exploiting task structure and runtime behavior. **Clustering-based Heuristics** group related tasks based on communication intensity or dependency proximity to reduce data transfer costs and improve data locality. **Duplication-based Heuristics** enhance parallelism by replicating selected tasks across multiple resources, minimizing wait times for dependent tasks and shortening the critical path. **Reactive rule-based Heuristics** adapt to runtime conditions using predefined decision rules, offering a lightweight form of adaptiveness without relying on learning or global optimization [71].

One of the most influential heuristic algorithms for DAG-based workflow scheduling is *HEFT* [6]. HEFT applies list scheduling by assigning priorities to tasks using upward

TABLE 2.2: SELECTED APPROACHES USING HEURISTICS

Algorithm	**Objectives**	**Category**	**Wo**
HEFT [6]	Makespan	List Scheduling	Yes
EHEFT/ECPOP [75]	Makespan, Energy	List Scheduling	Yes
MOHEFT [38]	Makespan, Energy	List Scheduling	Yes
MHRA [76]	Makespan, Energy	List Scheduling	Yes
Safari et al. [77]	Energy, SLA, Utilization	List Scheduling	Yes
RMREC [78]	Energy, Budget	List Scheduling + Greedy Heuristic	Yes
ECWS [79]	Cost, Energy, Utilization	List Scheduling	Yes
MM [48]	Energy, SLA	Greedy Heuristic	Yes
CMBFD [80]	Energy, Workload balance, SLA	Greedy Heuristic	Yes
ERAS-D [37]	Energy, CO_2, Resource Utilization, QoS	Greedy Heuristic	Yes
AGTI [81]	Energy, SLA constraints (Single obj.)	Greedy Heuristic	Yes
Khaleel et al. [82]	Makespan, Cost	Greedy Heuristic	Yes
FERPTS [83]	Runtime, Energy Cost	Greedy Heuristic	Yes
GeoDistributed [49]	Energy Cost (Single obj.)	Greedy Heuristic	Yes
REWS [84]	Energy, Reliability constraint (Single obj.)	Greedy Heuristic	Yes
CETSA [85]	Makespan, Energy, Cost (Merit function)	Greedy Heuristic	No
EPETS [62]	Makespan, Energy, Deadlines	Reactive Rule-based	No

Note. Wo = Workflow Oriented

ranking and allocating them to the resource that minimizes earliest finish time. While HEFT is widely recognized for its simplicity and effectiveness in minimizing makespan on heterogeneous systems, it does not consider energy consumption.

Several extensions of list-based approaches incorporate energy considerations. For example, *EHEFT* and *ECPOP* [75] introduce energy-aware extensions of traditional list-based heuristics that selectively power down underutilized processors and reschedule tasks to reduce idle energy consumption on heterogeneous cloud platforms. While these strategies deliver notable energy savings with limited impact on schedule length, their performance may degrade for highly parallel workflows where reducing the number of active processors constrains concurrency. Similarly, *MOHEFT* [38] extends HEFT into a multi-objective workflow scheduler capable of jointly optimizing makespan and energy consumption by maintaining a Pareto set of non-dominated scheduling alternatives. *MHRA* [76] later integrates multiple classic heuristics (LPT, SPT, LNS, LSTF) to reduce both makespan and energy consumption in heterogeneous clouds. *Safari et*

al. [77] incorporate DVFS-based deadline partitioning to improve utilization and reduce SLA violations by extending task execution times where slack exists while still meeting sub-deadline constraints. In recent literature, *RMREC* [78] introduces a two-phase energy-reduction and task-remapping strategy that operates within user budget limits to reduce excess energy usage during workflow execution. It demonstrates strong performance in cost-constrained environments; however, its effectiveness depends on accurate estimation of remaining budget and may degrade when cost fluctuations or runtime variability occur. *ECWS* [79] addresses the interplay between energy utilization, execution cost, and resource efficiency by filtering out low-effectiveness VMs and exploiting slack for reduced consumption. It increases utilization and lowers operational expense; however, performance depends on precise measurement of VM effectiveness, and may not directly incorporate stringent deadline or reliability requirements. Despite their benefits, these approaches rely on accurate prior estimates of task runtimes, workflow characteristics, and workload stability, which can limit effectiveness in environments with highly variable or unpredictable execution behavior.

In contrast, greedy scheduling approaches such as the *MM policy* [48] represent a shift toward energy-aware VM allocation by dynamically consolidating VMs based on resource utilization and minimizing the number of migrations required to preserve SLA guarantees. Its primary goal is to reduce power consumption under fluctuating demand; however, the use of CPU-centric thresholds and simplified migration-cost assumptions may lead to inefficiencies for communication- or memory-intensive workloads. Extending this direction, *CMBFD* [80] proposes a two-dimensional greedy heuristic for VM placement and migration that jointly considers CPU and memory utilization. By leveraging workload prediction and a double-threshold strategy, it improves consolidation efficiency while reducing SLA violations and migration overhead. Nonetheless, like MM, it focuses solely on VM-level decisions, without addressing the mapping of application tasks to VMs, and assumes homogeneous servers, which may limit performance in heterogeneous or network-intensive environments.

ERAS-D [37] extends beyond resource consolidation by integrating DVFS and workflow-structural characteristics into the decision process. ERAS-D estimates the earliest completion time on heterogeneous cloud resources, then applies frequency scaling to reduce CPU power usage without violating user deadlines. Finally, a backward layer-based scheduling strategy prioritizes critical path tasks and maximizes VM reuse to reduce idle time and operational overhead. However, ERAS-D assumes accurate performance prediction and the availability of discrete DVFS levels, and may incur suboptimal results when workload variability or network transfer delays dominate task execution costs. The *AGTI* [81] heuristic reduces overall energy consumption in DVFS-enabled environments by adaptively selecting task- or host-level frequency adjustments based on changes in both direct and idle energy while ensuring workflow deadlines are satisfied.

However, DVFS-aware heuristics must cope with a distinctly non-linear coupling between frequency scaling, execution time, and dependable QoS outcomes [37]. Empirical models show that while dynamic power drops with lower voltage/frequency, execution time expands proportionally, and the resulting longer makespan can amplify idle periods and indirect energy use across hosts [81]. In workflow settings with dependencies, this time expansion can propagate along the DAG, eroding slack and increasing the risk of missing user-specified deadlines, effectively translating energy decisions into SLA risk [81]. Recent deadline-constrained schedulers therefore avoid naïvely pushing frequencies to the minimum and instead scale within bounds derived from task slack and critical-path timing, selecting discrete levels that minimize total (direct + idle) energy while preserving feasibility of the deadline [37, 81]. Additionally, system-level studies note that energy and thermal behavior are tied to operational reliability, where reduced energy/heat can support longer Mean Time Between Failures (MTBF) ; thus, practical heuristics integrate timing feasibility, slack distribution, and energy models to ensure that DVFS gains are not offset by deadline violations or reliability degradation [37].

Khaleel et al. [82] incorporates multi-criteria priority evaluation to jointly minimize makespan and monetary cost, dynamically adapting scheduling decisions to fluctuations in VM performance and pricing in cloud settings. *FERPTS* [83] addresses SLA-driven energy efficiency by decomposing workflow scheduling into iterative provisioning, negotiation-based task placement, and runtime refinement steps, enabling more balanced resource utilization.

Recent greedy heuristics target emerging workflow requirements. *GeoDistributed scheduling* [49] focuses on workflows with geographically dispersed datasets and heterogeneous pricing models, optimizing energy cost using location- and time-aware decisions. While it accounts for network transmission time more comprehensively than prior methods, its model increases scheduling complexity and may rely on detailed price and bandwidth availability information that is not always observable. *REWS* [84] incorporates workflow reliability constraints by decomposing overall reliability into task-level targets, then applying an energy-efficient scheduling mechanism to satisfy both reliability and energy goals simultaneously. Although reducing replication overhead compared to many fault-tolerant approaches, the approach introduces additional reliability-prediction complexity and may require accurate failure modeling to prevent under-provisioning. Finally, *CETSA* [85] balances makespan, energy consumption, and budget using cost- and load-aware heuristics to prevent VM overloading while maintaining strong QoS adherence. However, CETSA addresses the scheduling of independent tasks rather than workflow-based applications and its multi-objective trade-offs are tuned by static threshold parameters, which may not fully adapt under rapidly shifting workload behavior or pricing conditions.

Reactive and rule-based heuristics offer lightweight and adaptive strategies suited for dynamic or uncertain cloud environments. *EPETS* [62] improves energy efficiency

while satisfying deadline constraints using a priority-guided two-stage scheduling and reassignment mechanism in heterogeneous cloud infrastructures. While beneficial for QoS-aware scheduling, it does not leverage workflow-oriented or dynamic execution feedback.

Overall, heuristic methods provide practical and scalable scheduling solutions for large-scale and real-time workflow environments. Their efficiency and lower complexity remain appealing; however, achieving strong performance in highly dynamic or multi-objective settings often requires more adaptive, learning-driven strategies.

2.3.0.3 Meta-Heuristic Approaches

Metaheuristic algorithms are inspired by natural and physical processes, and are designed to efficiently explore large, complex, and often non-convex solution spaces. Unlike traditional heuristics, which rely on fixed rules or deterministic logic, metaheuristics employ stochastic search mechanisms that help escape local optima and discover near-optimal solutions in high-dimensional scheduling problems [14].

A primary class of metaheuristics is **Swarm Intelligence Algorithms**, which simulate the collective behavior of decentralized systems. Particle Swarm Optimization (PSO) models a population of particles navigating the solution space by updating their positions based on individual and global best experiences, commonly targeting objectives such as makespan or cost. Similarly, Ant Colony Optimization (ACO) is inspired by the foraging behavior of ants, where artificial pheromones guide task assignments by reinforcing paths that lead to better solutions. In addition, Cat Swarm Optimization (CSO) mimics the resting (seeking mode) and chasing (tracing mode) behaviors of cats, balancing exploration and exploitation to enhance search efficiency in complex optimization problems [14].

Another major category includes **Evolutionary Algorithms (EA)** , which emulate the process of natural selection and evolution. Genetic Algorithms (GA) evolve a population of candidate schedules through genetic operators such as selection, crossover, and mutation, allowing the exploration of diverse scheduling strategies across generations. In addition, **Physics-inspired Algorithms** apply principles from physical systems to drive the search process. For example, Simulated Annealing mimics the annealing process in metallurgy, accepting worse solutions with a certain probability to escape local minima. The Gravitational Search Algorithm (GSA) models agents as masses that attract one another based on fitness, progressively converging towards promising regions of the solution space [14].

Furthermore, **Hybrid Metaheuristics** combine multiple strategies to exploit the complementary strengths of different algorithms, often improving convergence speed, solution quality, and robustness across diverse workflow scheduling scenarios [14].

PSO and its extensions have been extensively applied to workflow scheduling due to

TABLE 2.3: SELECTED APPROACHES USING META-HEURISTICS

Algorithm	**Objectives**	**Category**	**Wo**
Pandey et al. [86]	Makespan	PSO	Yes
DVFS-MODPSO [87]	Makespan, Cost, Energy	PSO	Yes
Chaotic PSO [88]	Makespan, Cost, Energy	PSO	Yes
HH-ECO [89]	Makespan, Energy	PSO	Yes
PSO-SA [90]	Makespan, Energy, Deadlines (Weighted function)	PSO	Yes
Greedy-Ant [91]	Makespan (Single obj.)	ACO	Yes
TETS [92]	Makespan, Energy	EA	Yes
EATTO [93]	Makespan, Energy, Throughput (Weighted function)	EA	Yes
HGALO-GOA [94]	Makespan, Cost, Energy, Throughput	EA	Yes
HGALO-SCA [95]	Makespan, Cost, Energy, Throughput	EA	Yes
HDSOS-GOA [96]	Makespan, Energy	EA	Yes
EMWSA [22]	Makespan, Energy	EA	Yes

Note. PSO = Particle Swarm Optimization, ACO = Ant Colony Optimization, EA = Evolutionary Algorithms, Wo = Workflow Oriented

their simplicity, adaptability, and effectiveness in exploring large solution spaces. The early work by *Pandey et al.* [86] employed PSO to minimize total execution and data transmission costs. Building on this, *DVFS-MODPSO* [87] jointly optimizes makespan, cost, and energy using dynamic voltage and frequency scaling. While effective in reducing power consumption, its multi-objective balance can introduce additional scheduling overhead. *Chaotic PSO* [88] further enhances exploration capability by incorporating chaotic search to avoid premature convergence and improve energy-efficiency and cost trade-offs, although the chaotic behavior may increase complexity in parameter tuning. *HH-ECO* [89], an approach extended from chaotic PSO, integrates adaptive mutation and heuristic migration to achieve significant reductions in makespan and energy, yet the model can struggle under rapidly shifting workload dynamics. Most recently, *PSO-SA* [90] combines PSO-based prioritization with simulated annealing-driven task allocation to jointly satisfy deadlines while minimizing energy and makespan, although improvements rely on carefully tuned hybridization to prevent additional runtime overhead.

ACO has been applied in workflow scheduling due to its distributed search capability and adaptability to complex computing environments. Although less frequently used in energy-aware scheduling, ACO variants such as *Greedy-Ant* [91] demonstrate its potential by applying an ant colony system to minimize makespan in DAG-based workflows, assuming a fully connected set of machines and non-preemptive task execution. The algorithm uses pheromone trails and heuristic enhancements to iteratively reinforce efficient task-to-resource mappings.

Evolutionary algorithms have been widely employed for multi-objective workflow scheduling due to their ability to discover diverse sets of Pareto-optimal solutions without requiring prior knowledge of the objective landscape. *NSGA-II* [97], a widely adopted genetic algorithm, preserves diversity using crowding distance and ensures convergence through elitism. Its successor, *NSGA-III* [98], extends this capability to many-objective problems using reference-point-based selection to maintain diversity. In addition to NSGA-II and NSGA-III, another prominent approach within the class of evolutionary algorithms is *MOEA/D* [99], which solves multi-objective problems through problem decomposition. Unlike population-ranking methods that operate on Pareto dominance, MOEA/D breaks the problem into a set of scalar subproblems and optimizes them simultaneously, promoting diversity and convergence through neighborhood-based collaboration. Although MOEA/D is not workflow-specific, its decomposition strategy and scalability make it a strong foundation for developing customized workflow scheduling algorithms, especially in cases with many conflicting objectives or constraints.

Building on these foundations, several evolutionary approaches for multi-objective workflow scheduling have been introduced in recent years. TETS [92] applies a two-phase GA with improved initialization to reduce makespan and energy utilization through DVFS-aware processor assignment. Although effective for small-scale workloads, its prioritization heuristics limit adaptability when workflow structures vary significantly. *EATTO* [93] uses a bat-inspired evolutionary strategy to jointly optimize makespan, energy, and throughput, yet its reliance on heuristic parameter tuning may degrade performance for highly heterogeneous resources. The hybrid *HGALO-GOA* [94] approach exploits ant-lion and grasshopper search behaviors with chaos-driven diversity, enabling strong trade-off handling among makespan, energy, cost, and throughput. However, the increased algorithmic complexity can introduce non-trivial overheads. To further enhance global exploration, *HGALO-SCA* [95] merges ant-lion and sine- cosine strategies to support four-objective optimization, yet its dependence on random chaos functions may lead to unstable scheduling quality on certain workflows. Meanwhile, *HDSOS-GOA* [96] targets energy-efficient scheduling using DVFS and HEFT-based ordering, but focuses primarily on fog/cloud edge scenarios and may not scale seamlessly to large IaaS clouds. Finally, *EMWSA* [22] incorporates workflow-structural knowledge through critical-task adjustment and idle-gap reuse, significantly improving energy and makespan optimization without excessive computational cost. A current limitation is that its strategies mainly target bi-objective cases, making generalization to richer QoS models an open opportunity for extension.

Despite their strong global search capabilities, metaheuristic approaches suffer from limited scalability when applied to large-scale scientific workflows. Since workflow scheduling is an NP-hard problem, these methods typically adopt an iterative, population-based framework in which a set of candidate solutions is repeatedly evalu-

ated over hundreds or even thousands of generations to reach convergence. As a result, computational cost grows rapidly with problem size. For example, the Greedy-Ant algorithm [91] has a rough time complexity of $O(iter_{max} \times K(nm + e))$, which scales linearly with both the maximum number of iterations and the number of ants. Similarly, the EATTO algorithm [93] incurs a complexity of $O(p \times N_{gen})$, highlighting that even when individual fitness evaluations are relatively efficient, the combined effect of population size and generation count leads to significant overhead. In addition, initialization techniques designed to enhance solution quality, such as HEFT-based initialization in DVFS-MODPSO [87] and Chaotic PSO [88], can introduce extra computational costs on the order of $O(N \times n^2 \times m)$, where n denotes the number of tasks and m the number of available resources. Consequently, the inherently iterative and population-driven nature of metaheuristics often results in high scheduling latency, limiting their practicality for real-time decision making in hyper-scale cloud environments with workflows comprising thousands of inter-dependent tasks.

While metaheuristic algorithms offer flexibility and effectiveness in handling complex, multi-objective workflow scheduling problems, they often rely on handcrafted representations and require extensive tuning. To overcome these limitations and better adapt to dynamic cloud environments, recent research has begun incorporating machine learning techniques into scheduling frameworks.

2.3.0.4 Machine Learning-based Approaches

Machine learning-based scheduling has gained significant traction in recent years, offering the potential to learn optimal or near-optimal scheduling policies by leveraging historical data, online feedback, or direct interaction with the environment. These approaches are particularly valuable in complex, dynamic, and uncertain cloud environments, where traditional rule-based or search-based methods may struggle to generalize or adapt.

One of the most widely explored paradigms in intelligent scheduling is **Reinforcement Learning (RL)** , where the workflow scheduling problem is formulated as a Markov Decision Process (MDP) . In this setting, an agent interacts with the environment by observing system states, selecting scheduling actions, and receiving feedback in the form of rewards. Over time, the agent learns a policy that maps states to actions in order to maximize long-term cumulative rewards. DRL extends traditional RL by incorporating deep neural networks to approximate value functions or policies, allowing learning in high-dimensional and continuous state or action spaces. DRL methods have demonstrated strong adaptability and generalization capabilities in complex cloud environments, although they typically require significant training time and careful design of the reward function.

A critical challenge in applying conventional DRL to large-scale workflow schedul-

TABLE 2.4: REWARD FUNCTION DESIGN CATEGORIES IN ML-BASED SCHEDULING

Design	**Key Characteristics & Limitations**	**Representative Works**
Linear Combination	Aggregates multiple conflicting objectives (e.g., makespan, energy, cost) into a single scalar reward using static or dynamic weights. Easy to implement but highly sensitive to weight tuning and unable to discover non-convex Pareto-optimal solutions.	DQTS [15], DRLB-TSA [16], DeepRM+ [100], ADRL [64], ETAWSDRL [40]
Hierarchical	Decomposes reward across multiple levels (e.g., global vs. local agents) or applies lexicographic objective prioritization. Enhances scalability and modular learning, but may introduce communication overhead and rigid trade-off handling.	Hierarchical DRL [101], MARL [3], MOPWSDRL [102]
Vector-valued	Maintains separate value functions per objective or learns multiple policies to approximate the Pareto front. Enables explicit trade-off analysis and dynamic preference adaptation, but increases computational complexity and training instability.	Emerging MO-RL trends [9]; Rare in current cloud workflow literature

ing is the state explosion problem [8]. As the number of tasks and dependencies in a DAG increases, the dimensionality of the state space grows exponentially, making it difficult for standard neural networks with fixed-size input layers to effectively represent the system state [8]. To address this, recent studies have adopted advanced embedding techniques. For instance, Recurrent Neural Networks (RNNs) and Long Short-Term Memory (LSTM) networks have been employed to process variable-length sequences of task features, enabling the model to capture temporal dependencies and sequential decision patterns [101, 103].

Beyond RL, other machine learning paradigms have also been applied to workflow

scheduling. **Supervised Learning** models can be trained on historical data to predict task execution times, resource availability, or optimal task-resource mappings. These models enhance scheduling efficiency by enabling more informed decisions without relying on exhaustive search. In parallel, **Unsupervised Learning** techniques like K-means and hierarchical clustering have been employed to group similar tasks or resources, reduce problem dimensionality, and estimate task-resource affinities in large-scale systems.

TABLE 2.5: SELECTED APPROACHES USING MACHINE LEARNING

Algorithm	**Objectives**	**Category**	**Wo**
ML-PowerScheduling [104]	SLA, Power cost	ML + ILP	Yes
ARLCA [39]	Energy, SLA	RL	No
DQTS [15]	Makespan, Load Balance	DRL	Yes
DRLB-TSA [16]	Makespan, SLA, Energy	DRL	No
DeepRM [23]	Average Job Slowdown	DRL	No
DeepRM+ [100]	Turnaround Time, Cycling Time	DRL	No
ADRL [64]	SLA, Utilization	DRL	No
DRL-Cloud [105]	Makespan, Energy, Task Rejection Rate	DRL	Yes
MOPWSDRL [102]	SLA, Power cost	DRL	No
DRL-WorkloadScheduler [106]	Energy, Latency	DRL	No
LowCarbon [32]	Energy, QoS	DRL	Yes
ODTS [107]	Energy	DRL	No
DeepEnergyJSV2.0 [24]	Energy	DRL	Yes
ETAWSDRL [40]	Energy, Cooling Cost	DRL	Yes
Hierarchical DRL [101]	Energy, Latency	DRL	No
Jayanetti et al. [52]	Makespan, Energy	DRL	Yes
MARL [3]	Energy, Renewable Energy	DRL	Yes
IDRQN [103]	Task Offloading	DRL	Yes
DRL + FL [108]	Energy, Delay, Load Balance	DRL	No

Note. ILP = Integer Linear Programming, RL = Reinforcement Learning, DRL = Deep Reinforcement Learning, Wo = Workflow Oriented

Machine learning techniques are increasingly used in cloud task and workflow scheduling to enhance decision-making and improve energy efficiency. *ML-Power-Scheduling* [104] leverages machine learning to predict CPU usage and response time, enabling profit-aware and power-efficient scheduling decisions in managed data-centers. A key limitation is that exact ILP optimization becomes computationally expensive and struggles to scale to large, real-world scheduling scenarios, as noted in their evaluation. RL is also widely adopted in cloud environments for dynamic resource management under uncertainty. *ARLCA* [39] introduces an advanced reinforcement learning-based

VM consolidation agent that improves energy efficiency while significantly reducing SLA violations by learning optimal placement policies under dynamic workloads. However, its focus solely on CPU utilization means other critical resources like memory and network bandwidth are not considered, which can still lead to occasional service violations as noted in the results.

DRL has been actively explored for workflow scheduling, particularly in scenarios involving DAG-based task models, SLA constraints, and multi-resource environments. *DQTS* [15] employs Deep Q-Learning with experience replay to jointly minimize makespan and improve load balancing in DAG-based workflows, using entropy-based reward fusion. Similarly, *DRLB-TSA* [16] applies Deep Q-Learning to minimize makespan, SLA violations, and energy consumption in large-scale cloud environments. *DeepRM+* [100] advances the DRL scheduling paradigm by using policy gradient methods with imitation learning to minimize average weighted turnaround time and cycling time. It processes image-based representations of system states and learns from shortest-job-first heuristics, enabling robust job scheduling in stochastic multi-resource clusters. To improve QoS under dynamic web workloads, *ADRL* [64] integrates anomaly-aware decision-making into Deep Q-Learning, using isolation forest to detect workload anomalies and applying a two-level vertical and horizontal scaling mechanism that balances SLA adherence and resource adjustment overhead. In another line of work, *DRL-Cloud* [105] adopts a semi-Markov Decision Process framework with two-stage decision-making for energy-efficient workflow scheduling and resource provisioning, accounting for heterogeneous VMs, hard deadlines, and dynamic electricity pricing. *MOPWSDRL* [102] first assigns priorities to workflows based on dependency complexity and to VMs based on electricity cost, then uses a DQN scheduler to minimize makespan and energy consumption during dynamic scheduling. However, results depend strongly on accurate priority estimation, and the model may face adaptation challenges in unpredictable workload surges. *DRL-WorkloadScheduler* [106] uses Deep Q-Networks to analyze workload properties (CPU, memory, execution time) and schedules tasks to improve energy efficiency while maintaining QoS.

With growing concerns about energy efficiency and sustainability in cloud computing, recent DRL-based approaches have focused on optimizing environmental and operational metrics such as energy consumption, carbon footprint, and electricity cost. *LowCarbon* [32] formulates the workflow scheduling problem using DRL to jointly minimize carbon emissions, energy cost, and QoS penalties. *ODTS* [107] introduces an online DRL framework that dynamically assigns and migrates tasks to reduce energy consumption, using a dual-agent scheduling and migration strategy with preemption. Complementing these efforts, *DeepEnergyJSV2.0* [24] uses a policy-gradient method (PPO) to optimize task scheduling targeting energy minimization across hybrid workloads. *ETAWSDRL* [40] calculates task and datacenter priorities based on computational demand and temperature while using DQN to reduce both energy and cooling costs.

However, considering temperature adds complexity; requires continuous thermal monitoring infrastructure, which may not be available in all datacenters.

To better manage the complexity of large-scale and geo-distributed cloud environments, recent studies have adopted advanced DRL architectures such as hierarchical and multi-agent frameworks. *Hierarchical DRL* [101] introduces a two-tier control system where a global agent based on deep Q-learning handles VM allocation, while local controllers use LSTM-enhanced reinforcement learning to manage power consumption at individual servers. This layered approach improves both latency and energy efficiency. Similarly, *Jayanetti et al.* [52] has a hierarchical action design that helps differentiate between edge and cloud nodes, improving decisions for energy- and time-sensitive IoT workflows. The hybrid actor-critic model with PPO reduces energy use and execution time while satisfying more workflow deadlines. Extending this idea to a distributed multi-cloud setting, *MARL* [3] proposes a hierarchical multi-agent actor-critic framework to minimize brown energy consumption and task makespan in datacenters powered by a mix of renewable and non-renewable energy. The model trains local and global agents under partially observable Markov decision processes using centralized training with decentralized execution. Agents coordinate scheduling decisions based on green energy availability, workload characteristics, and server heterogeneity, making it highly adaptable to real-world, energy-aware cloud scenarios. *IDRQN* [103] further expands on partial observability using LSTM-based DQN for mobile edge computing with fine-grained offloading. Additionally, *DRL+FL* [108] combines double DQN with federated learning for privacy-aware distributed training in edge-cloud resource allocation scenarios.

2.4 Pareto Optimality & Evaluation Metrics

In multi-objective optimization, researchers are often concerned with finding a set of trade-off solutions rather than a single optimal solution. These trade-offs form what is known as the Pareto front, where no solution is strictly better than another across all objectives. Because of this, traditional scalar performance metrics are inadequate. Instead, MO researchers rely on specialized performance indicators that can capture the quality of a solution set in terms of its convergence to the true Pareto front, the diversity of solutions, and how well the front is covered [30]. These metrics allow for meaningful comparison across algorithms and problem instances.

Evaluating the quality of solution sets is essential since multiple trade-offs must be considered simultaneously. Performance indicators are used to assess various aspects such as convergence to the true Pareto front, diversity of solutions, and dominance relations between sets. Commonly used metrics are mentioned in the Table 2.6.

In particular, when evaluating the trade-off between green objectives (e.g., energy consumption, CO_2 emissions) and performance objectives (e.g., makespan, reliability),

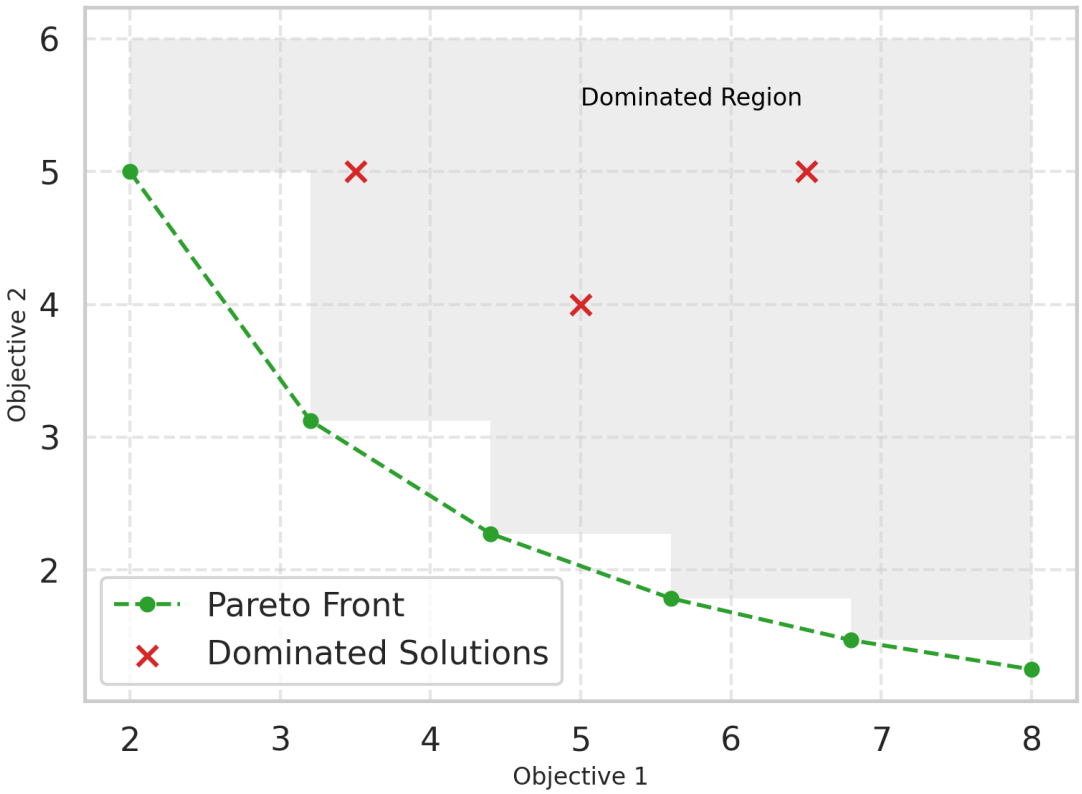

Fig. 2.5: Pareto Front Visualization

it is crucial to select metrics that capture both the quality of individual solutions and the breadth of the trade-off front. Metrics such as Hypervolume and IGD are particularly well-suited for this purpose as they simultaneously assess convergence (minimizing energy and maximizing performance) and diversity (providing a range of options from high-performance/high-energy to low-performance/low-energy) [30].

2.4.1 Evaluation Metrics of Related Studies

Evaluating scheduling algorithms across the literature reveals that most works prioritize conventional performance measures tailored to the specific goals of cloud workflow execution, typically focusing on minimizing time, energy, or monetary cost. As shown in Tables 2.7, 2.8, and 2.9, a majority of heuristic and meta-heuristic approaches employ scalar metrics such as makespan, resource utilization, energy consumption, or SLA violation rates. These metrics provide straightforward and interpretable results but inherently capture only a single objective at a time or aggregate multiple objectives into a single value, potentially masking trade-offs. Only a smaller subset of studies adopt Pareto-based evaluation, allowing the assessment of both convergence and diversity of solutions. Such analyses provide a more comprehensive and unbiased view of algorithmic performance in multi-objective scheduling scenarios, yet remain underutilized

TABLE 2.6: SUMMARY OF PERFORMANCE METRICS IN MULTI-OBJECTIVE OPTIMIZATION

Metric	**Description**	**Conv.**	**Div.**
Generational Distance	Measures the average distance from each point in the approximated Pareto front to the nearest point on the true Pareto front.	✓	
Inverted Generational Distance	Measures the average distance from each point on the true Pareto front to the nearest point in the approximation.	✓	✓
Maximum Pareto Front Error	Captures the largest distance between a point in the approximation and the true Pareto front.	✓	
Spacing	Evaluates the uniformity of distances between neighboring solutions in the approximation.		✓
Δ-index	Considers both the spread and uniformity of the Pareto front approximation.		✓
Entropy	Measures how evenly the solutions are distributed along the front.		✓
C-metric	Computes the fraction of solutions in one set that are dominated by another set.	✓	
ε-indicator	Determines the minimum factor (additive or multiplicative) by which one solution set must be adjusted to dominate another.	✓	
Hypervolume	Calculates the volume in objective space dominated by the approximation, bounded by a reference point.	✓	✓

Note. Conv. = Convergence, Div. = Diversity

despite their importance in decision-making for heterogeneous and dynamic cloud environments.

While Pareto front-based evaluation is somewhat common in multi-objective cloud workflow scheduling when non-energy-aware literature is also considered [21, 34, 109, 110], it still remains uncommon in DRL-based scheduling approaches. Few researches, such as DQTS [15] considers the Pareto front — and even then, only a single solution

TABLE 2.7: SUMMARY OF EVALUATION METRICS OF SELECTED HEURISTIC STUDIES

Algorithm	Pareto	Evaluation Metrics
MOHEFT [38]	Yes	Makespan and Energy Consumption shown as Pareto fronts
MHRA [76]	No	Energy consumption, Cloud elasticity, Scheduling overhead
Safari et al. [77]	No	Resource Utilization, Average Execution Time, Average Energy Consumption/Energy Savings, SLA Violation
RMREC [78]	No	Total Energy Consumption, Execution Cost
ECWS [79]	No	Energy consumption, Cost, Resource Utilization, Deadline Satisfaction
MM [48]	No	Total energy consumption, SLA violation percentage, Number of VM migrations, Average SLA violation
CMBFD [80]	No	Total energy consumption, SLA violation percentage, Number of VM migrations, Resource utilization, Scalability
ERAS-D [37]	No	Makespan, Energy consumption, Energy cost, CO_2 emission, Provider profit, Resource utilization rate
FERPTS [83]	No	Run time, Energy cost
EPETS [62]	No	Energy consumption, Total execution time / makespan, Deadline violations, Number of task failures, Resource utilization

Note. Pareto = Has done Pareto analysis

from it is utilized. This is likely because DRL approaches generally focus on learning a single optimal policy, aligned with a scalar reward function, rather than generating a set of policies representing trade-offs. As noted by Hayes et al. [9], this scalarisation-centric design can obscure the underlying decision trade-offs, limit flexibility when preferences shift, and hinder explainability. Their work emphasizes the benefits of explicitly modeling multiple objectives in reinforcement learning through frameworks like Pareto optimality and coverage sets. This remains an open and promising direction for future research in DRL-based scheduling, particularly for applications that demand transparent trade-offs and dynamic preference adaptation.

2.5 Experimental Datasets and Simulations

2.5.1 Real-World Datasets

The domain of energy-efficient multi-objective cloud workflow scheduling currently lacks a good, standardized benchmark dataset specifically designed to cover all necessary criteria (e.g., energy consumption, cost, makespan, and realistic workload/hardware

TABLE 2.8: SUMMARY OF EVALUATION METRICS OF SELECTED META-HEURISTIC STUDIES

Algorithm	**Pareto**	**Evaluation Metrics**
DVFS-MODPSO [87]	Yes	Non-dominated Solutions, Makespan, Cost, Energy Consumption
Chaotic PSO [88]	Yes	Makespan, Cost, Energy consumption
HH-ECO [89]	No	Makespan, Degree of Disparity, Average Energy Consumption, SLA Violations, Total Cost, Migration Efficiency, Waiting Time Ratio
TETS [92]	Yes	Makespan, Energy consumption, Percentage and number of Pareto-optimal solutions, Improvement rates vs baseline algorithms
HGALO-GOA [94]	Yes	Inverted Generational Distance, Coverage Ratio, Maximum Spread, Spacing
HGALO-SCA [95]	Yes	Inverted Generational Distance, Coverage Ratio, Maximum Spread, Spacing
HDSOS-GOA [96]	Yes	Fitness value, Energy consumption, Resource utilization ratio, Performance improvement vs other algorithms
EMWSA [22]	Yes	Hypervolume, Convergence, Diversity

Note. Pareto = Has done Pareto analysis

TABLE 2.9: SUMMARY OF EVALUATION METRICS OF SELECTED REINFORCEMENT LEARNING STUDIES

Algorithm	**Pareto**	**Evaluation Metrics**
ARLCA [39]	No	Energy Consumption, VM Migrations, SLA Violations
MOPWSDRL [102]	No	Makespan, Energy Consumption
ETAWSDRL [40]	No	Makespan, Energy Consumption, Resource Utilization, Scheduling Overhead
Jayanetti et al. [52]	No	Energy consumption, Makespan, Percentage of deadline hits, Percentage of jobs completed
MARL [3]	No	Total energy consumption, Makespan, Total green energy surplus

Note. Pareto = Has done Pareto analysis

heterogeneity). As a result, different studies often rely on different datasets and evaluation methodologies, which makes direct, fair comparison of proposed scheduling techniques challenging [8]. Developing a comprehensive, multi-objective-focused benchmark is therefore a significant future avenue for research.

2.5.1.1 Pegasus Workflow Structures Dataset

The Pegasus Workflow Structures Dataset [36] provides a widely adopted benchmark suite for evaluating scientific workflow scheduling algorithms in cloud and grid environments. These workflows are modeled as DAGs, where nodes represent individual computational tasks and edges denote data or control dependencies between them. Pegasus workflows are derived from real scientific applications across diverse domains, capturing the computational and data-flow characteristics typically encountered in large-scale research projects. As shown in Table 2.10, the dataset includes several workflows that vary in terms of size, computational complexity, and I/O intensity. Among RL-based approaches, DQTS [15], MOPWSDRL [102], ETAWSDRL [40], Jayanetti et al. [52], and MARL [3] employ DAGs from this dataset for performance evaluation.

TABLE 2.10: WORKFLOWS IN PEGASUS WORKFLOW STRUCTURE

Algorithm	**Description**	**Workflow Type**	**Application Area**
Montage	A workflow used to create an image mosaic of sky	Data Intensive	Astronomy
Cybershake	A workflow used to depict the earthquake threats in an area with the help of the Probabilistic Seismic Hazard Analysis method	Computational	Earthquake
Epigenomics	A workflow interprets the processing of genetic data to implement the different genome sequencing operations	Computational, Data Intensive	Genetic data
SIPHT	A workflow that implements the bioinformatics problems	Computational	Bioinformatics
LIGO	A workflow used for the identification of gravitational waves and to examine the data attained from compact binary systems	Data Intensive	Gravitational works

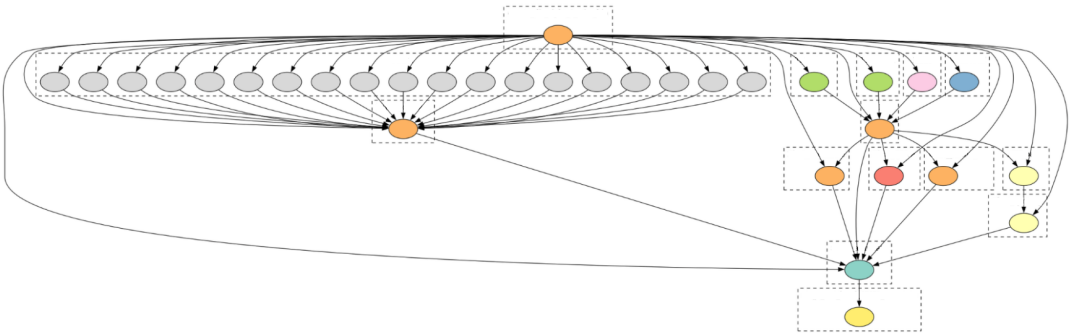

Fig. 2.6: SIPHT Workflow Example (from Pegasus)

2.5.1.2 Real-World Cluster Trace Datasets

Several real-world cluster trace datasets have been released by major cloud service providers to support research in large-scale workflow and resource scheduling [28, 29, 111]. These datasets capture realistic workload patterns, resource utilization, and scheduling behaviors in production environments.

The Google Cluster Trace V3 [28] offers a full trace of workloads executed on eight Google Borg compute clusters throughout the month of May 2019. It captures detailed data on job submissions, task execution, machine state transitions, and scheduling decisions, making it a valuable resource for studying task characteristics, VM characteristics, cluster-level scheduling, and utilization trends. Several scheduling approaches such as DRL-Cloud [105], ODTS [107], and IDRQN [103] have employed this dataset for evaluation purposes.

The Alibaba Cluster Trace Program 2018 [29] provides workload traces from approximately 4000 machines over a period of 8 days. It includes detailed information on production batch jobs, notably with DAG structures that describe task dependencies. This dataset is used for evaluating DAG-based scheduling and resource optimization techniques. Approaches such as DeepRM+ [100], LowCarbon [32], and DeepEnergyJSV2 [24] have utilized this dataset for performance evaluation.

The Azure Public Dataset V2 [111] contains a representative subset of first-party VM workloads from one geographical Azure region. It includes anonymized telemetry data that reflects VM lifecycles, resource allocations, and usage patterns, supporting research in VM-level scheduling, forecasting, and anomaly detection.

2.5.1.3 Real-World Server Specification Datasets

The Standard Performance Evaluation Corporation (SPEC) provides a widely recognized repository of benchmark results that reflect detailed specifications and performance metrics of real-world server hardware [47]. These benchmarks include information on processor models, core counts, memory configurations, clock speeds,

and power consumption, along with standardized performance scores across diverse workloads.

The SPEC dataset is frequently used by researchers to derive realistic infrastructure configurations for simulations and performance modeling, particularly in cloud and distributed computing studies. Among its benchmarks, the SPECpower_ssj2008 [47] suite is especially notable for evaluating server energy efficiency trade-offs under varying workload intensities. This benchmark has been utilized by RL-based approaches such as MARL [3] and IDRQN [103] to model realistic energy-performance trade-offs in server environments.

2.5.2 Simulation Frameworks

Most research on cloud task/workflow scheduling has leveraged CloudSim [19], a widely used, open-source Java-based simulation toolkit designed for modeling and evaluating cloud environments. CloudSim provides a flexible and extensible platform for simulating cloud datacenters, resource management strategies, and scheduling algorithms under diverse workload conditions.

The CloudSim simulation model captures the multi-layered structure of a cloud computing environment, incorporating workflow models, task dependencies, and dynamic resource characteristics. It includes core components such as datacenters, VMs, and cloudlets (tasks), where each workflow can be represented as a DAG. The Cloud Information Service maintains a registry of available resources across datacenters, tracking VM configurations and real-time states [19].

Task scheduling is managed by the Datacenter Broker, which acts as an intermediary between user-submitted workflows and available resources. The broker applies the implemented scheduling algorithm to dynamically assign tasks to VMs based on availability and suitability. Tasks are allocated to VMs distributed across multiple hosts, effectively simulating real-world scenarios where resources are shared and heterogeneous [19].

CloudSim allows customization of system parameters, including the number of hosts, VM configurations, and energy models. Workflow tasks are characterized by computational workloads (measured in Million Instructions), memory requirements, and inter-task dependencies, while VMs can be configured with specific CPU capacities, RAM sizes, and energy consumption profiles [19]. This versatility makes CloudSim a powerful tool for evaluating the performance of scheduling strategies under realistic and reproducible conditions.

In addition to CloudSim, several other simulation tools have been utilized in cloud workflow scheduling research. WorkflowSim [121] is an extension of CloudSim specifically developed to support large-scale workflow simulations. It introduces native support for DAG-based workflows and includes modules for clustering and resource

TABLE 2.11: SIMULATION TOOL USAGE AMONG RELATED LITERATURE

Simulation Tool	Related Works
CloudSim	Prob [74], ICPCP [65], EIPR [112], CETSA [85], SSPPH [113], Dyna [114], Rodriguez et al. [115], BPSO [72], HPSO [116], Barrett et al. [66], ADRL [64], MARL [3]
WorkflowSim	MOPWSDRL [102], ETAWSDRL [40], HH-ECO [89], ICFWS [68], STSDD [117]
iFogSim	HDSOS-GOA [96], SSSPSO [118], IDRQN [103]
jMetal	FDHEFT [21], Calzarossa et al. [119]
SimGrid	MW-HBDCS [120]

provisioning. However, it is no longer actively maintained. jMetal [122] is a Java-based framework tailored for solving MOO problems using metaheuristics, such as genetic algorithms and particle swarm optimization. While not a simulator itself, it is frequently integrated into custom simulation environments for evaluating optimization-based scheduling algorithms. iFogSim [123] focuses on modeling and simulating resource management in IoT, edge, and fog computing environments. Although it provides energy-aware simulation capabilities, its relevance to cloud-centric workflow scheduling is limited. CloudSim Plus [124] is an extended, modular, and object-oriented version of CloudSim that introduces additional utilities and improvements. Finally, SimGrid [125] is a toolkit designed for simulating distributed systems at scale, offering detailed network and energy modeling features. It has been employed in grid and heterogeneous datacenter scheduling studies but is less commonly used in workflow-specific simulations.

2.5.3 Comparison Algorithms in Reinforcement Learning Approaches

As shown in Table 2.12, most reinforcement learning-based workflow scheduling approaches evaluate their performance against simple heuristic baselines such as First-Come-First-Serve (FCFS) , Round-Robin, Min-Min, and Greedy strategies. A few works include comparatively stronger or context-specific heuristics/meta-heuristics, such as Tetris [126], PSO, ACO, and FERPTS [83]. However, these are still heuristic-based methods. Despite the increasing sophistication of DRL approaches, their evaluation often remains limited to these heuristic baselines. This highlights a gap in benchmarking against more advanced or hybrid scheduling techniques.

2.6 Comparison Tables of Related Studies

Tables 2.13, 2.14, and 2.15 present a comparative analysis of RL-based cloud task and workflow scheduling algorithms discussed in Section 2.3.0.4. These tables summarize

TABLE 2.12: COMPARISON ALGORITHMS IN RL-BASED WORKFLOW SCHEDULING APPROACHES

Approach	**Comparison Algorithms (Excl. Variants)**
DQTS [15]	FCFS, Round-Robin, Min-Min, Max-Min, Minimum Completion Time
DRL-Cloud [105]	Round-Robin, Greedy, FERPTS [83]
LowCarbon [32]	FCFS, Tetris [126], Ideal MPC (ILP)
ODTS [107]	Best Fit, Round-Robin, Median Absolute Deviation
DeepEnergyJSV2.0 [24]	First Fit, Random, Tetris [126]
Liu et al. [101]	Round-Robin
MOPWSDRL [102]	HEFT [6], ACO, CSO
ETAWSDRL [40]	PSO, ACO, CSO
Jayanetti et al. [52]	Random, Energy Efficient Scheduling Heuristic, Energy and Delay Aware Heuristic, EFT (HEFT variant)
MARL [3]	Random, Green-Opt

key aspects of each approach, including the problem factors considered, their main contributions, and the limitations observed in existing work.

In addition, Table 2.16 provides a structured feature-wise comparison of the same algorithms using binary indicators. Each row corresponds to an algorithm, while each column represents the presence or absence of a specific capability or characteristic, such as support for DAG-based workflow scheduling, handling dynamic scenarios, optimization objectives (e.g., makespan, energy consumption, QoS/SLA), simulation environments like CloudSim, usage of real-world inspired datasets, evaluation using Pareto-based metrics, and architectural aspects like hierarchical scheduling, GNNs, or the use of PPO for training. Green checkmarks (✓) and red crosses (✗) are used to visually indicate whether each feature is addressed by a given approach.

2.7 Graph Neural Networks for Cloud Scheduling

The basis for using GNNs for the cloud scheduling problem (Relating to **RQ2**) stems from its similarities to the Flexible Job Shop Scheduling Problem (FJSSP) , a variation of the Job Shop Scheduling Problem found in Operational Research literature. The FJSSP shares several conceptual similarities with cloud workflow scheduling. Both domains involve assigning a set of interdependent tasks or jobs to a finite pool of resources (machines in FJSSP, VMs in cloud scheduling) while adhering to operational constraints. In FJSSP, each job comprises a sequence of operations that can be processed by different machines, mirroring cloud scenarios where tasks are executable on multiple VMs depending on resource availability and compatibility.

Constraints are central to both problems: FJSSP must consider machine capabilities and scheduling restrictions, much like cloud systems must account for VM attributes

TABLE 2.13: COMPARISON OF EXISTING RL-BASED APPROACHES

Related Work	Factors Considered	Advantages	Limitations
DQTS [15]	• DAG Workflows • Makespan and load balance optimization • Dynamically arriving tasks	• A scheduling scheme in the cloud computing environment using deep Q-learning • Pareto plots for trade-offs • Pegasus datasets utilized for evaluations	• Energy consumption not considered • Simple algorithms considered for comparison: FCFS, RR, MINMIN, MAXMIN, MCT, DataAware
DRLB-TSA [16]	• Makespan and energy consumption optimization with SLA constraints • Dynamically arriving tasks	• DQN-based multi-metric optimization • Compared with MOABCQ and RATS-HM (MO algos)	• No DAG workflows • Lacks pareto-based evaluations
DeepRM+ [100]	• Independent tasks • Turnaround Time and cycling time optimization • Job arrival modeled as a Poisson process	• Deep RL with CNN and Imitation Learning (SJF as expert) • Alibaba cluster trace 2018 utilized for evaluations	• No DAG workflows • Energy consumption not considered • Not multi-objective • Lacks pareto-based evaluations
ADRL [64]	• Resource scaling algorithm • SLA and response time optimization • Anomaly handling	• Deep Q-Learning with anomaly-aware triggering • Combining vertical and horizontal scaling decisions	• No DAG workflows • Makespan and energy consumption not considered • Lacks pareto-based evaluations

TABLE 2.14: COMPARISON OF EXISTING RL-BASED APPROACHES CONT.

Related Work	Factors Considered	Advantages	Limitations
DRL-Cloud [105]	• DAG workflows • Makespan, energy consumption, and task rejection rate optimization • Changing user request patterns	• Two-stage Deep Q-Learning system using SMDP formulation • Target network and experience replay • Google Cluster workload traces utilized	• Makespan not considered • Lacks pareto-based evaluations • Compared only with simple baselines: RR, greedy, energy-aware
LowCarbon [32]	• DAG Workflows • Energy consumption and task delay optimization • Random job arrival and fluctuating energy prices	• DRL-based policy with policy gradient training • Uses reward scaling and baseline function • Evaluated using Alibaba traces	• No DAG workflows • Makespan not considered • Lacks pareto-based evaluations
ODTS [107]	• Independent tasks • Energy consumption optimization with SLA constraints • Dynamically arriving tasks	• Online Deep Q-learning based Task Scheduling with dual agents (assignment + migration) • Google Cluster workload traces utilized	• No DAG workflows • Makespan not considered • Lacks pareto-based evaluations
DeepEnergy JSV2.0 [24]	• DAG Workflows • Energy consumption optimization	• Utilizes PPO with importance sampling for training the agent • Uses Alibaba Cluster 2018 dataset	• Makespan not considered • Lacks pareto-based evaluations

TABLE 2.15: COMPARISON OF EXISTING RL-BASED APPROACHES CONT.

Related Work	Factors Considered	Advantages	Limitations
Liu et al. [101]	• Independent tasks • Power, latency, and SLA optimization	• Hierarchical DRL framework with two-tier system: global VM allocation + local power management • Pegasus Dataset utilized for evaluations	• No DAG workflows • Makespan not considered • Lacks pareto-based evaluations
MARL [3]	• DAG workflows • Energy consumption and makespan optimization • Renewable energy usage optimization • Workflow arrival modeled as a Poisson process	• Multi-Agent Actor-Critic • Hierarchical scheduling with global and local agents • Pegasus Dataset utilized for evaluations	• Lacks pareto-based evaluations • Only compared with simple algorithms (Random, Green-Opt heuristic) and variations of proposed method
IDRQN [103]	• Energy consumption, cost, latency, and network usage optimization • Mobile Edge Computing setup • DAG workflows (partial)	• Improved Deep Recurrent Q-Network with LSTM + Candidate Network Set • Uses Google Cluster Trace and VR gaming subtasks for simulations	• Not fully cloud-workflow oriented • Lacks pareto-based evaluations
DRL + FL [108]	• Independent tasks • Energy consumption, delay, and load balance optimization • Job arrival modeled as a Bernoulli process	• Double Deep Q-Learning for resource allocation • Combined with Federated Learning	• No DAG workflows • Makespan not explicitly considered • Lacks pareto-based evaluations

TABLE 2.16: FEATURE COMPARISON OF RL-BASED SCHEDULING ALGORITHMS

Algorithm/ Method	Workflow Scheduling (DAG)	Dynamic Scenarios	Makespan Objective	Energy Consumption Objective	QoS/SLA Objective	CloudSim for Simulations	Real Data Inspired Datasets	Pareto-Based Comparisons	Compare with MO Algos	GNNs Utilized	PPO for Training	Multi-Stage/Hierarchical Scheduling
DQTS [15]	✓	✓	✓	✗	✗	✓	✓	✓	✗	✗	✗	✗
DRLB-TSA [16]	✗	✓	✓	✓	✓	✓	✓	✗	✓	✗	✗	✗
DeepRM [23]	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
DeepRM+ [100]	✗	✓	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗
ADRL [64]	✗	✓	✗	✗	✓	✓	✓	✗	✗	✗	✗	✓
DRL-Cloud [105]	✓	✓	✓	✓	✓	✗	✓	✗	✗	✗	✗	✓
LowCarbon [32]	✓	✓	✗	✓	✓	✗	✓	✗	✗	✗	✗	✗
ODTS [107]	✗	✓	✗	✓	✓	✗	✓	✗	✗	✗	✗	✓
DeepEnergyJSV2.0 [24]	✓	✓	✗	✓	✗	✗	✓	✗	✗	✗	✓	✗
Liu et al. [101]	✗	✓	✗	✓	✓	✗	✓	✗	✗	✗	✗	✓
MARL [3]	✓	✓	✓	✓	✗	✓	✓	✗	✗	✗	✗	✓
IDRQN [103]	✓	✓	✗	✓	✓	✗	✓	✗	✗	✗	✗	✗
DRL + FL [108]	✗	✓	✗	✓	✓	✗	✓	✗	✗	✗	✗	✗
Proposed	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

such as memory, storage, and bandwidth. However, a critical distinction lies in the complexity of task relationships: cloud workflows are often represented by DAGs, capturing intricate dependencies, whereas FJSSP typically involves simpler precedence relationships.

Recent research demonstrates that GNN-based approaches have achieved significant success in FJSSP. For instance, [56] introduced a multi-action GNN framework that outperformed existing scheduling strategies, while [25] proposed a hierarchical scheduling approach for the Dynamic FJSSP (DFJSSP) , using GNNs at the lower level to enhance dynamic task allocation under random arrivals. These successes highlight GNNs' capability to model complex task structures and resource mappings effectively.

Despite these advances, GNN integration in cloud workflow scheduling remains largely unexplored. Existing DRL-based schedulers for cloud workflows often overlook the rich structural information encoded in DAGs, potentially missing opportunities to enhance scheduling accuracy and adaptability. The successes in FJSSP suggest that leveraging GNNs within DRL frameworks could significantly strengthen cloud scheduling systems by enabling better modeling of task dependencies, handling dynamic environments, and balancing multi-objective requirements such as makespan, energy efficiency, and QoS.

This gap presents a clear and promising research direction: adapting GNN-enhanced DRL strategies from FJSSP to the more complex and dynamic context of cloud workflow scheduling.

2.8 Popular DAG based Workflow Engines

Contemporary research on cloud workflow scheduling is closely aligned with the evolution of industrial orchestration platforms responsible for coordinating large-scale distributed applications. Whereas academic studies predominantly emphasize optimization objectives such as makespan reduction, cost minimization, or energy efficiency, production-grade systems tend to foreground operational concerns including reliability, observability, and robust fault tolerance. In practice, many industry-adopted orchestration engines employ DAG abstractions, which offer a structurally coherent model that bridges theoretical scheduling approaches with real-world execution requirements.

2.8.1 Apache Airflow

Apache Airflow [17] is one of the most widely adopted open-source workflow orchestration platforms used in both industry and research environments. Airflow models workflows as DAGs, where nodes represent tasks and edges define execution dependencies. Workflows are defined programmatically using Python, enabling dynamic pipeline generation and flexible scheduling policies.

Airflow provides features such as task retries, dependency management, monitoring dashboards, and distributed execution through worker queues. Its scheduler continuously evaluates task readiness based on dependency completion and execution state, making it suitable for batch workflows, data engineering pipelines, and scientific processing tasks. Although Airflow includes scheduling capabilities, its primary focus is orchestration rather than optimization; therefore, it typically relies on external resource managers or infrastructure-level schedulers for resource allocation decisions.

2.8.2 Prefect

Prefect [127] is a modern workflow orchestration framework designed to address limitations observed in earlier systems such as Airflow, particularly in terms of dynamic execution and observability. Prefect adopts a "dataflow-first" design philosophy, allowing workflows to adapt at runtime rather than relying solely on static DAG definitions.

Unlike traditional schedulers, Prefect emphasizes resilience and state management through centralized orchestration services that track task execution states, failures, and retries. It supports hybrid execution across local machines, cloud environments, and containerized infrastructure. Prefect's architecture enables dynamic branching, parameterized workflows, and event-driven execution, making it well suited for machine learning pipelines and adaptive cloud workloads.

2.8.3 Kubernetes as a Workflow Execution Platform

Although Kubernetes [128] is primarily a container orchestration system rather than a workflow engine, it has become a foundational platform for workflow execution in cloud-native environments. Kubernetes manages container deployment, scaling, fault tolerance, and resource allocation across clusters of machines, effectively acting as the infrastructure-level scheduler beneath many workflow systems.

Workflow engines such as Argo Workflows, Kubeflow Pipelines, and Prefect deployments frequently run on Kubernetes clusters, leveraging its built-in scheduling mechanisms. Kubernetes schedules workloads based on resource availability, constraints, and policies, enabling efficient utilization of heterogeneous computing resources.

From a scheduling perspective, Kubernetes introduces an additional abstraction layer where workflow-level scheduling decisions interact with container-level resource scheduling. This separation highlights an important distinction between workflow orchestration (task dependency management), and infrastructure scheduling (resource placement and scaling).

Recent research has explored the integration of learning-based decision mechanisms into existing orchestration frameworks to improve resource utilization and energy efficiency. One such approach is **RLKube** [129], which extends the native Kubernetes scheduler through a RL-based plugin.

RLKube integrates with the Kubernetes scheduling framework by introducing a custom component during the PreScore phase of scheduling. At this stage, cluster state information is collected using Prometheus, including metrics related to node utilization and resource availability. These metrics are forwarded to a Double Deep Q-Network (DDQN) model that evaluates candidate nodes and assigns a score representing their suitability for executing a given pod. However, RLKube operates only at the node-selection level. The overall scheduling workflow remains governed by Kubernetes, meaning that pod lifecycle management and final placement enforcement are still

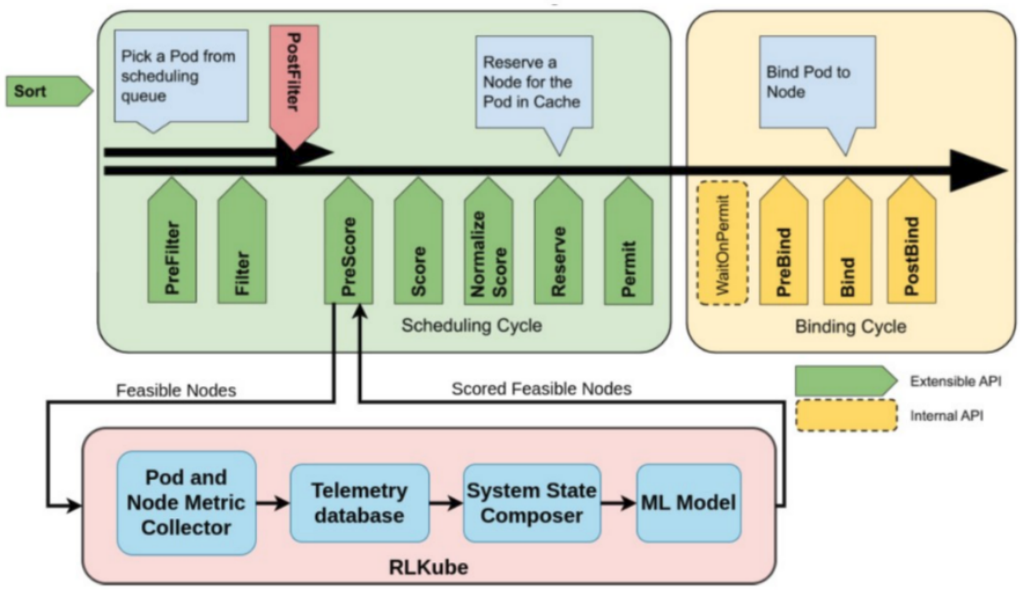

Fig. 2.7: Overview of RLKube Approach

Note. Taken from RLKube [129]

handled by the native scheduler.

DRL-FORCH [130] is a scalable orchestration framework that employs DRL to optimize service deployment across heterogeneous fog nodes while satisfying QoS constraints. DRL-FORCH models orchestration as a sequential decision-making problem in which deployment actions must balance multiple competing objectives, including latency, energy consumption, and scalability. The orchestrator is implemented using a Deep Set (DS) neural network architecture, enabling permutation-invariant processing of variable-sized node sets. To improve learning stability and efficiency, the framework incorporates invalid action masking, preventing the agent from selecting infeasible deployment decisions during training and inference.

2.9 Limitations of Current Research

Despite the growing interest in applying DRL to cloud task and workflow scheduling, several limitations continue to constrain the effectiveness, generalizability, and practical relevance of current research efforts.

One of the most prominent gaps lies in the limited use of DRL for true multi-objective optimization. While many studies apply DRL to optimize individual objectives such as makespan or energy consumption, only a handful attempt to balance multiple conflicting objectives in a principled manner. In most cases, multiple objectives are combined into a single scalar reward using weighted sums, which masks the inherent

trade-offs and undermines the exploration of Pareto-optimal solutions. The lack of Pareto-awareness restricts these models from effectively navigating the complex design space of real-world cloud scheduling problems, where optimizing for performance, energy, and QoS simultaneously is crucial.

Another significant limitation is the sparse integration of workflow-specific structures, particularly DAGs, which are fundamental to modeling task dependencies in scientific workflows. Many existing DRL-based approaches assume independent tasks or use flat queues, neglecting the precedence constraints and communication dependencies that define real-world workflows. This simplification not only reduces the realism of evaluations but also limits the applicability of proposed solutions in practical workflow management scenarios.

Compounding this issue is the underutilization of GNNs, despite their strong suitability for capturing the structural relationships within DAG-based workflows. GNNs offer a powerful way to model task dependencies and graph-based features, yet their integration with DRL for scheduling remains largely unexplored. This represents a missed opportunity to improve the structural awareness and generalization ability of scheduling agents in dynamic and complex environments.

Evaluation practices in the literature also reveal notable shortcomings. Many DRL-based schedulers are benchmarked against simplistic baselines such as Random, Round Robin, or Greedy heuristics, with little effort to compare against established multi-objective optimization algorithms like MOHEFT, NSGA-II, or SPEA2. Furthermore, Pareto-based evaluation metrics such as Hypervolume or IGD are rarely used, hindering the ability to assess trade-off quality and the true multi-objective performance of these methods.

Finally, there is limited emphasis on the practical applicability and system integration of DRL-based scheduling algorithms. Most studies are confined to simulation environments and fail to consider how these methods could be deployed in real-world systems. There is a lack of research on integrating DRL agents with existing workflow management tools (e.g., Pegasus, Airflow) or cloud orchestration platforms (e.g., Kubernetes), which raises concerns about the transferability and scalability of the proposed solutions. As a result, much of the current work remains theoretical, with limited pathways toward production-level adoption.

CHAPTER 3

METHODOLOGY

3.1 Methodology Overview

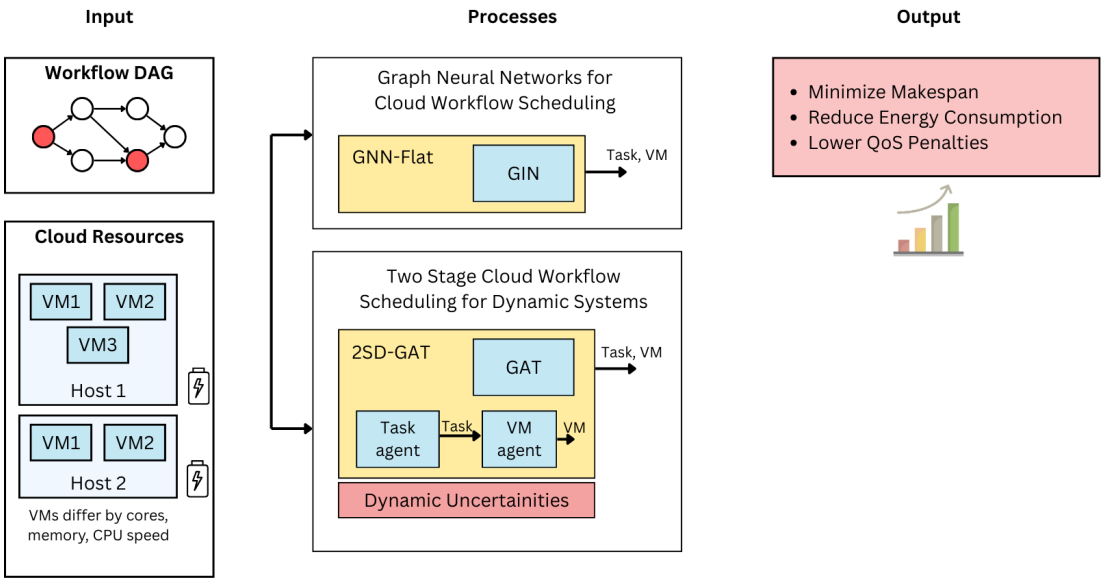

Fig. 3.1: Graphical Abstract of the Research Methodology

Figure 3.1 provides a graphical abstract of the research methodology proposed in this study. The system accepts workflows represented as DAGs and a virtualized cloud environment as input. Two primary approaches are explored: (1) a Graph Neural Network-based scheduler that leverages DAG structure to learn task-VM mappings via a single-stage actor-critic PPO framework; and (2) a Two-Stage Reinforcement Learning-based scheduler that employs a two-stage decision process, separately selecting tasks and VMs through coordination. Both models are trained using synthetic datasets generated under varying conditions, with environment simulations conducted using CloudSim. The final outputs include optimized scheduling decisions that minimize makespan, energy consumption, and latency-based QoS penalties, along with Pareto fronts for multi-objective evaluation.

In the last sections, several implementations of pluggable schedulers/integrations are developed to evaluate the proposed scheduling algorithms in real-world-inspired environments (addressing **RQ4**). These include (1) an Apache Airflow scheduler integration, (2) a Kubernetes-based scheduler integration, and a (3) a standalone event-driven scheduling tool with DAG execution capabilities.

This chapter covers the full methodological pipeline developed to realize energy-efficient workflow scheduling in cloud systems using deep reinforcement learning. The

following contributions are addressed in detail:

- **Solution Model for Energy-Efficient Cloud Task Scheduling:** The formalization of a multi-objective optimization problem targeting makespan, energy consumption, and QoS sensitivity under DAG constraints.
- **Graph Neural Networks for Cloud Workflow Scheduling:** Design and training of a GNN-based scheduler using a PPO-based reinforcement learning framework capable of capturing workflow structure and VM heterogeneity.
- **Two-Stage Cloud Workflow Scheduling for Dynamic Systems:** Introduction of a two-stage PPO-based reinforcement learning agent, incorporating GAT-based feature extraction and preference-based reward shaping to support Pareto-optimal scheduling.
- **Simulation and Evaluation:** A CloudSim-based simulation framework for evaluation under varying dataset sizes, workload distributions, and comparative baselines including heuristic and evolutionary algorithms.

It should be noted that Section 3.2 presents the first stage of this research: **GNN-Flat**, the Graph Neural Network-based scheduler formulated as a flat-action agent. This model integrates a novel GIN-based architecture to effectively encode workflow and resource characteristics. Section 3.3 extends this foundation by introducing **2SD-GAT**, a two-stage reinforcement learning approach that significantly enhances scheduling performance. This advanced model incorporates dynamic system uncertainties and evaluates multi-objective trade-offs using a GAT-based agent hierarchy. Together, these two stages represent the progressive refinement of the proposed scheduling methodology.

3.2 Graph Neural Networks for Cloud Workflow Scheduling

Figure 3.2 illustrates the architectural overview of the DRL methodology used in GNN-Flat. Initially, datasets are generated based on various parametric configurations. A simulated environment serves as the foundation for the training process, wherein the agent interacts with the environment by executing actions and receiving corresponding rewards along with state information and environmental features. Through this feedback mechanism, the agent learns to optimize the dual objectives: minimizing makespan and reducing energy consumption.

3.2.1 Problem Modeling

In this study, workflows are represented as DAGs, which are commonly used to model dependencies between tasks [105]. A DAG is an acyclic graph $G = (V, E)$, where V is

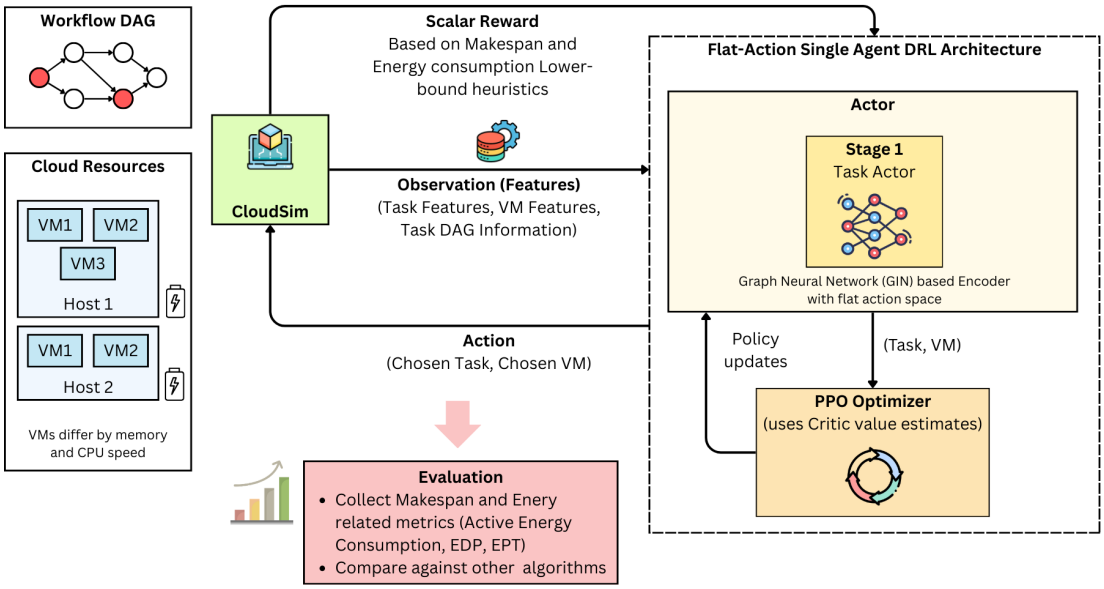

Fig. 3.2: RL methodology used for the proposed algorithm

the set of vertices (tasks) and $E \subseteq V \times V$ is the set of directed edges representing dependencies between tasks. For any directed edge $(u, v) \in E$, task u must be completed before task v can start.

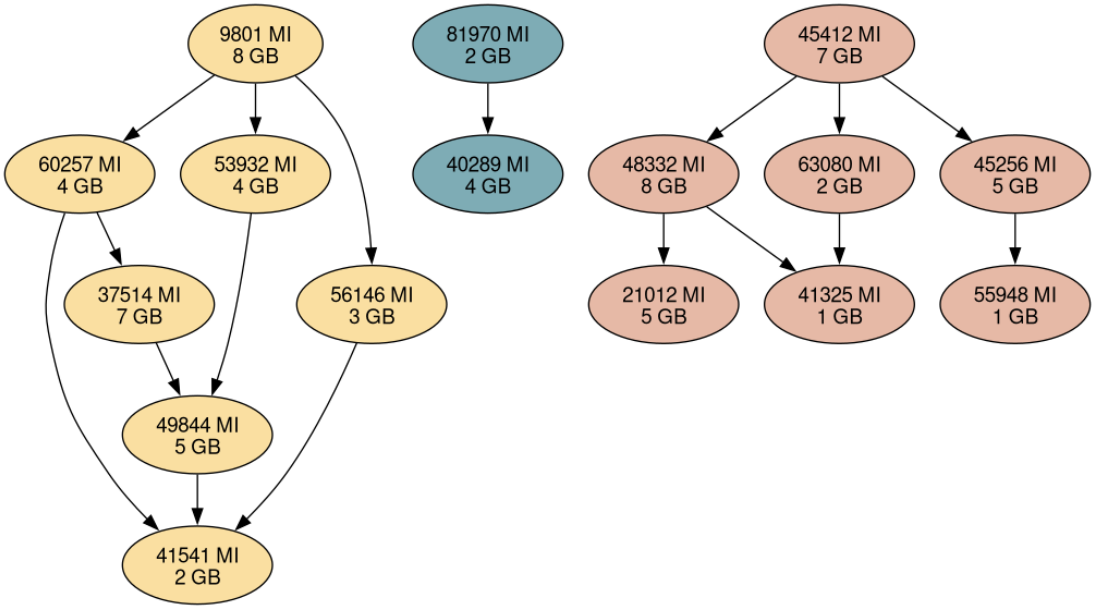

Fig. 3.3: Sample workflows represented as DAGs

Note. Tasks are annotated with their computational requirement (in Million Instructions MI) and memory requirement (in Gigabytes GB).

Makespan is a key metric in scheduling workflows, representing the total time required to complete all tasks in a workflow. Given a DAG $G = (V, E)$ representing

the workflow, the makespan M of the workflow is defined as in eq. (3.1).

$$M = \max_{J_i \in V} t_{J_i}^{\text{end}} \tag{3.1}$$

Here, $t_{J_i}^{\text{end}}$ denotes the completion time of task $J_i \in V$. The value of M_{J_i} can be derived as shown in eq. (3.2).

$$st_{J_i}^{\text{end}} = t_{J_i}^{\text{start}} + T_{J_i,\text{VM}_k} \tag{3.2}$$

In this expression, $t_{J_i}^{\text{start}}$ denotes the time at which the scheduler assigns task J_i to virtual machine VM_k, and T_{J_i,VM_k} represents the execution time of task J_i on VM_k. For a scheduling decision to be valid, the task must be compatible with the selected VM. This compatibility is determined based on the resource requirements specified by the user — for instance, a task may require a VM with certain memory capacity, number of cores, or other hardware characteristics. These constraints are treated as part of the user's resource requirements and must be strictly enforced by the scheduler to ensure service compliance. Hence, the assignment of tasks is restricted to VMs that satisfy the corresponding compatibility criteria.

In this section (Section 3.2), compatibility is determined by the memory requirements of the task and the VM. A task J_i can only be assigned to VM_j if $R_{J_i} \leq R_{\text{VM}_j}$, where R_{J_i} is the memory required by task J_i, and R_{VM_j} is the memory allocated to VM_j. When this condition is satisfied, the execution time T_{J_i,VM_j} can be calculated using eq. (3.3).

$$T_{J_i,\text{VM}_j} = \frac{L_{J_i}}{S_{\text{VM}_j}} \tag{3.3}$$

Here, L_{J_i} is the length of task J_i, and S_{VM_j} is the CPU speed of VM_j. It is assumed that the VM operates at full utilization with a constant CPU speed S_{VM_j} during the execution of the task.

Total energy consumption is a critical metric in cloud scheduling, representing the cumulative energy used by all hosts in a data center over a given period. The total energy consumption, denoted as E^{total}, is defined as in eq. (3.4).

$$E^{\text{total}} = \int_0^T P^{\text{total}}(t)\, dt \tag{3.4}$$

Here, $P^{\text{total}}(t)$ represents the total power consumption of the data center at time t. It is calculated as the sum of the power consumption of all N^H hosts, as shown in eq. (3.5).

$$P^{\text{total}}(t) = \sum_{i=1}^{N^H} P_{H_i}(t) \tag{3.5}$$

The power consumption $P_{H_i}(t)$ of host H_i at time t is determined by its utilization $U_{H_i}(t)$, which varies between the thresholds $U_{H_i}^{\text{low}}$ and $U_{H_i}^{\text{high}}$. The corresponding power values at these thresholds are $P_{H_i}^{\text{low}}$ and $P_{H_i}^{\text{high}}$. Assuming a linear relationship between utilization and power, $P_{H_i}(t)$ can be expressed as in eq. (3.6).

$$P_{H_i}(t) = P_{H_i}^{\text{low}} + (P_{H_i}^{\text{high}} - P_{H_i}^{\text{low}}) \cdot \frac{U_{H_i}(t) - U_{H_i}^{\text{low}}}{U_{H_i}^{\text{high}} - U_{H_i}^{\text{low}}} \tag{3.6}$$

For simplicity, using the power at idle ($P_{H_i}^{\text{idle}}$) and peak utilization ($P_{H_i}^{\text{peak}}$), the above equation can be reduced to eq. (3.7).

$$P_{H_i}(t) = P_{H_i}^{\text{idle}} + (P_{H_i}^{\text{peak}} - P_{H_i}^{\text{idle}}) \cdot U_{H_i}(t) \tag{3.7}$$

The utilization $U_{H_i}(t)$ of host H_i is influenced by the VMs running tasks on it. Let $\text{VM}_{H_i,t}^{\text{active}}$ represent the active VMs on host H_i at time t. The utilization $U_{H_i}(t)$ is given by eq. (3.8).

$$U_{H_i}(t) = \sum_{\text{VM}_j \in \text{VM}_{H_i,t}^{\text{active}}} \frac{S_{\text{VM}_j}}{S_{H_i}} \tag{3.8}$$

Here, S_{VM_j} is the CPU speed of VM_j, and S_{H_i} is the CPU speed of host H_i.

The objective is to find an optimal scheduling policy that minimizes both makespan (M) and total energy consumption (E^{total}). This multi-objective optimization problem is formulated as shown in eq. (3.9).

$$\text{minimize } \left(M, E^{\text{total}}\right) \tag{3.9}$$

This optimization is subject to two main constraints: (1) each task must satisfy memory requirements, such that $R_{J_i} \leq R_{\text{VM}_j}$, and (2) task precedence constraints defined by the DAG structure must be strictly respected.

3.2.2 Lower Bound Heuristics

In this study, lower-bound heuristics provide baseline estimations for makespan and energy consumption, which are later used to formulate the reward function (addressing **RQ3**). These lower bounds serve as optimistic estimates, ensuring that the actual makespan and energy consumption will always be greater than or equal to these values.

The makespan lower-bound estimation at time t is calculated by determining the maximum estimated completion time among all tasks in the workflow DAG. For each task J_i, the completion time lower-bound depends on its scheduling status. If a task is scheduled, its actual completion time is used. For unscheduled tasks, the completion time lower-bound is computed by evaluating all compatible VMs and considering both parent task availability and the VM's readiness.

Formally, the makespan lower-bound estimation, denoted as M_t^{est}, is defined as shown in eq. (3.10).

$$M_t^{\text{est}} = \max_{J_i \in V} M_{J_i,t}^{\text{est}} \tag{3.10}$$

Here, $M_{J_i,t}^{\text{est}}$ represents the estimated completion time lower bound of task J_i. This is recursively derived using eq. (3.11).

$$M_{J_i,t}^{\text{est}} = \begin{cases} M_{J_i} & \text{if } J_i \text{ is scheduled,} \\ \min_{\text{VM}_j \in \text{VM}_{J_i}^{\text{cmp}}} \left(\max_{J_k \in J_{J_i}^{\text{par}}} \left(M_{J_k,t}^{\text{est}}, M_{\text{VM}_j,t} \right) + T_{J_i,\text{VM}_j} \right) & \text{otherwise.} \end{cases} \tag{3.11}$$

In this equation $\text{VM}_{J_i}^{\text{cmp}}$ represents the set of compatible VMs for task J_i, $J_{J_i}^{\text{par}}$ denotes the parent tasks of J_i, and $M_{\text{VM}_j,t}$ is the current finish time of VM_j, as defined in eq. (3.12).

$$M_{\text{VM}_j,t} = \max_{J_k \in J_{\text{VM}_j,t}^{sch}} M_{J_k} \tag{3.12}$$

Here, $J_{\text{VM}_j,t}^{sch}$ refers to all the tasks scheduled on VM_j before time t, and M_{J_k} is the completion time of task J_k, as defined in eq. (3.2).

The active energy consumption lower-bound estimation at time t is computed using the energy consumption explicitly caused by running jobs (disregarding the idle power consumption). If a task is scheduled, its active energy consumption is based on the task's execution time and the assigned VM. For unscheduled tasks, the active energy consumption is estimated as the minimum possible energy cost across compatible VMs.

Formally, the active energy consumption lower-bound estimation, denoted as E_t^{est}, is defined in eq. (3.13).

$$E_t^{\text{est}} = \sum_{J_i \in V} E_{J_i,t}^{\text{est}} \tag{3.13}$$

Here, V represents all the tasks, and $E_{J_i,t}^{\text{est}}$ is the estimated active energy cost of task J_i. This cost is defined in eq. (3.14).

$$E_{J_i,t}^{\text{est}} = \begin{cases} E_{J_i,\text{VM}_k}, & \text{if } J_i \text{ is already scheduled to } \text{VM}_k, \\ \min_{\text{VM}_j \in \text{VM}_{J_i}^{\text{cmp}}} \left(E_{J_i,\text{VM}_j} \right), & \text{otherwise.} \end{cases} \tag{3.14}$$

In this expression, E_{J_i,VM_j} is the active energy consumed when task J_i is scheduled on VM_j. It is calculated using eq. (3.15).

$$E_{J_i,\text{VM}_j} = T_{J_i,\text{VM}_j} \cdot P_{\text{VM}_j} \tag{3.15}$$

Here, T_{J_i,VM_j} is the execution time of task J_i on VM_j, and P_{VM_j} is the power consumption of VM_j. The power consumption P_{VM_j} is derived as shown in eq. (3.16) using eq. (3.8) and the active component of eq. (3.7).

$$P_{\text{VM}_j} = (P_{H_k}^{\text{peak}} - P_{H_k}^{\text{idle}}) \cdot \frac{S_{\text{VM}_j}}{S_{H_k}} \tag{3.16}$$

Here, $P_{H_k}^{\text{peak}}$ and $P_{H_k}^{\text{idle}}$ represent the peak and idle power consumption of the host H_k where VM_j is located. The term S_{VM_j} is the CPU speed of VM_j, and S_{H_k} is the CPU speed of the host H_k.

3.2.3 Data Preprocessing

To ensure faster convergence of the model, the input data is preprocessed into a structured format suitable for the reinforcement learning framework. The preprocessing step handles task-level, VM-level, and pairwise information, as well as the task dependency graph. Input feature information is given in table 3.1.

Additionally, since a workflow may contain multiple start and end nodes, the algorithm adds a common start node and end node consisting of dummy tasks (with 0 MI length and compatibility with all VMs). This is helpful in simplifying the workflows.

3.2.4 Reinforcement Learning Framework

RL is a paradigm in machine learning in which an agent interacts with an environment to learn optimal actions through trial and error. The agent receives feedback in the form of rewards, which guide its learning to maximize cumulative rewards over time.

- State (s): Represents the environment's current condition. The state provides essential information for the agent to make decisions. Let S_t denote the state at time t.
- Action (a): An action is a choice made by the agent in a given state. Let A_t denote the action taken by the agent at time t.
- Reward (r): A scalar feedback signal provided by the environment after each action, indicating the immediate benefit of the action. The goal is to maximize the cumulative reward over time. Let R_{t+1} be the reward received after taking action A_t in state S_t.

TABLE 3.1: SUMMARY OF INPUT FEATURES

Feature		Name	Description
$F^{\text{ts}}_{i,t}$	$I^{\text{sch}}_{J_i,t}$	Task scheduled flag	A binary flag indicating whether the task has already been scheduled.
$F^{\text{tr}}_{i,t}$	$I^{\text{ready}}_{J_i,t}$	Task ready flag	A binary flag denoting whether the task is ready to be scheduled, determined by checking if all its parent tasks are scheduled.
$F^{\text{tc}}_{i,t}$	$M^{\text{est}}_{J_i,t}$	Task completion time	The estimated completion time for unscheduled tasks, or the actual completion time for tasks that are already scheduled. (Equation 3.11)
$F^{\text{vmc}}_{i,t}$	$M_{\text{VM}_i,t}$	VM completion time	The time at which the VM will become available for scheduling new tasks, based on the currently scheduled tasks. (Equation 3.12)
$F^{\text{vms}}_{i,t}$	$\frac{1}{S_{\text{VM}_i,t}}$	VM CPU speed	The inverse of CPU speed of the VM in MIPS. The inverse is used to improve the model convergence.
$F^{\text{vme}}_{i,t}$	$\frac{P^{\text{peak}}_{H_k} - P^{\text{idle}}_{H_k}}{S_{H_k}}$	VM active energy consumption rate	The active energy consumption of a VM, calculated using the host characteristics (H_k being the host of VM_i).
$A_{i,j,t}$	$I^{\text{cmp}}_{J_i,\text{VM}_j,t}$	Compatibility flag	A binary flag indicating whether the task can be executed on the VM, based on the task's resource requirements and the VM's specifications ($R_{\text{VM}_j} \geq R_{J_i}$).
$B_{i,j,t}$	$I^{\text{dep}}_{J_i,J_j,t}$	Task dependency flag	A binary flag representing whether there is a dependency (directed edge) from task J_i to task J_j. If a task is scheduled to a VM, a new dependency is added between the previous task of the VM and the scheduled task.

- Policy (π): Defines the agent's behavior by mapping states to probabilities of selecting each possible action. The policy can be deterministic ($\pi(s) = a$) or stochastic ($\pi(a|s) = P(A_t = a|S_t = s)$).

- Value Function ($V(s)$): Estimates the expected cumulative reward an agent can obtain from a given state s, following a policy π. The value function is defined as in eq. (3.17) [131] where $\gamma \in [0, 1)$ is the discount factor, which balances

immediate and future rewards.

$$V^{\pi}(s) = \mathbb{E}_{\pi}\left[\sum_{k=0}^{\infty} \gamma^{k} R_{t+k+1} \middle| S_t = s\right] \tag{3.17}$$

- Action-Value Function ($Q(s, a)$): Estimates the expected cumulative reward of taking an action a in state s and then following policy π. The action-value function is defined as in eq. (3.18) [131].

$$Q^{\pi}(s, a) = \mathbb{E}_{\pi}\left[\sum_{k=0}^{\infty} \gamma^{k} R_{t+k+1} \middle| S_t = s, A_t = a\right] \tag{3.18}$$

In DRL literature, PPO is a popular algorithm designed to improve the stability and efficiency of policy updates in continuous and discrete action spaces [132]. PPO optimizes a policy by iteratively updating it using a clipped surrogate objective, which prevents large policy updates and ensures stable learning. The objective function in PPO is defined as in eq. (3.19) [132].

$$L^{\text{PPO}}(\theta) = \mathbb{E}_t\left[\min\left(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\right)\right] \tag{3.19}$$

In eq. (3.19):

- $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$ is the probability ratio of the new policy π_θ to the old policy $\pi_{\theta_{\text{old}}}$.
- $\hat{A}_t$ is the advantage estimate at time t, calculated as $\hat{A}_t = Q(s_t, a_t) - V(s_t)$.
- ϵ is a hyperparameter that controls the range for clipping, limiting the extent of policy updates.

In GNN-Flat, PPO is applied to the cloud workflow scheduling problem, where the agent's goal is to minimize both the makespan and energy consumption. The reward R_t is designed to reflect the multi-objective optimization framework's focus, penalizing the agent for longer makespans and higher energy usage. This dual-objective reward structure ensures that the agent learns to balance performance and sustainability in cloud environments effectively.

Initially, reward functions based solely on makespan were considered, such as the immediate reward $R_{t+1} = -\left(M_{t+1}^{\text{est}} - M_t^{\text{est}}\right)$, with a terminal reward reflecting the final makespan. While this formulation helped the agent focus on execution time, it ignored energy considerations.

To incorporate energy consumption, a straightforward extension was attempted:

$$R_{t+1} = -\left(M_{t+1}^{\text{est}} - M_t^{\text{est}}\right) - \left(E_{t+1}^{\text{est}} - E_t^{\text{est}}\right)$$

However, this version was found to perform poorly due to the scale mismatch between makespan and energy consumption. Since these metrics operate on different numerical ranges, the agent's learning process was dominated by whichever signal had a larger magnitude.

To resolve this, a normalization strategy was adopted, where the deltas were scaled by their respective current estimates. This led to the final reward function shown in eq. (3.20). The parameters α and β allow adjustment of the importance of each component, but were empirically set to $\alpha = 1$ and $\beta = 1$ during training.

$$R_{t+1} = -\alpha \left(\frac{M^{\text{est}}_{t+1} - M^{\text{est}}_{t}}{M^{\text{est}}_{t+1}} \right) - \beta \left(\frac{E^{\text{est}}_{t+1} - E^{\text{est}}_{t}}{E^{\text{est}}_{t+1}} \right) \tag{3.20}$$

3.2.5 Agent Architecture and Model Design

The agent is designed to optimize task scheduling by encoding task and VM features using a GNN architecture. The model comprises several components, including encoders for tasks and VMs pairs, followed by graph construction and processing through GIN layers. The architecture is illustrated in Figure 3.4.

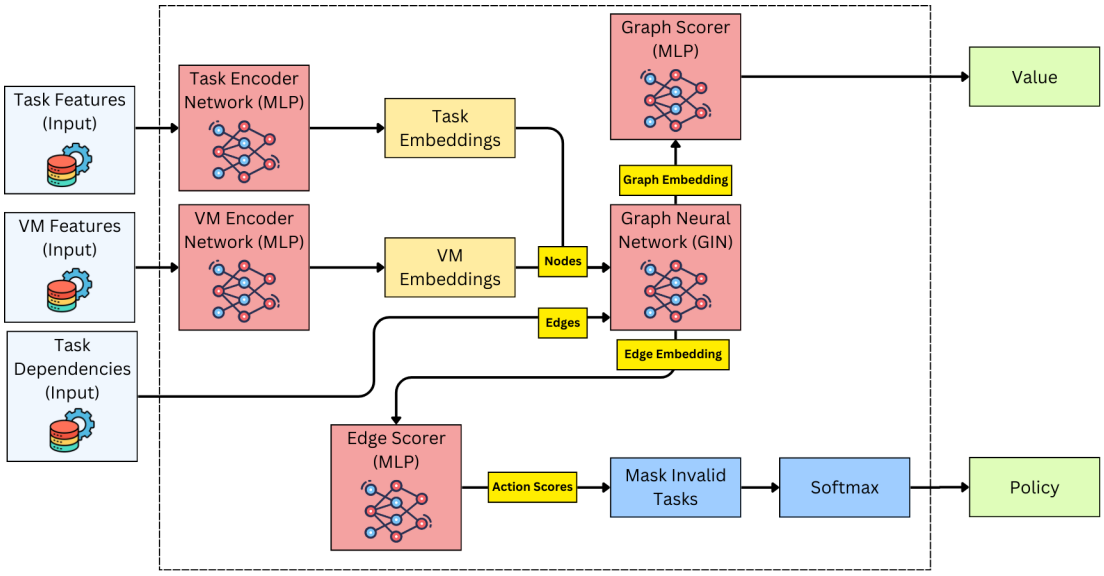

Fig. 3.4: Agent architecture used for the proposed algorithm

Each task is represented by a feature vector containing three elements: the task scheduled flag, the task ready flag, and the task completion time. These features are processed by the task encoder to generate task embeddings. Similarly, each VM is characterized by a three feature, the VM finish time, VM CPU speed, and VM active energy consumption rate, and is processed by the VM encoder to generate VM embeddings.

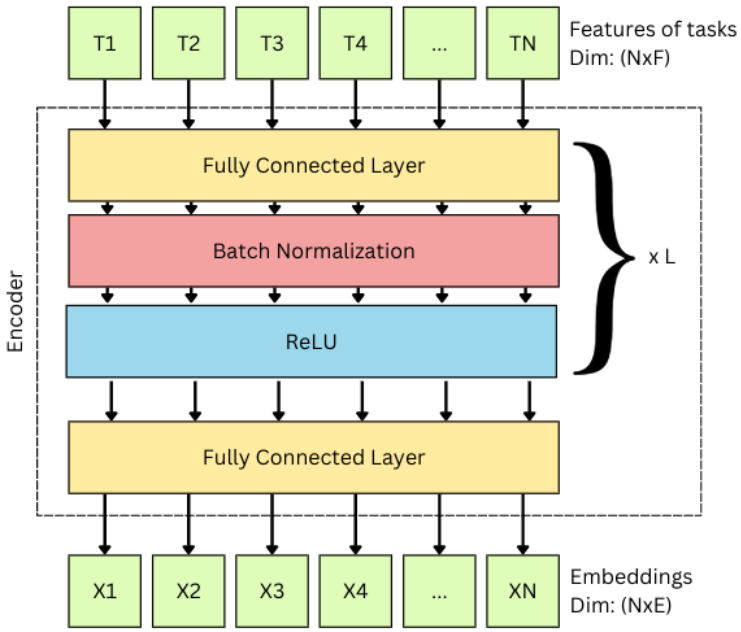

Fig. 3.5: Structure of the encoder

The primary objective of these encoders is to transform task and VM features into a unified embedding space, enabling graph construction. Each encoder follows the same structure, as illustrated in Figure 3.5. The encoder is implemented as a Multi-Layer Perceptron (MLP) with batch normalization and ReLU activation. The MLP transformation for a single layer is defined as in eq. (3.21) where x is the input feature vector, $W^{(l)}$ and $b^{(l)}$ are the weights and biases of the l-th layer, and $z^{(l)}$ is the pre-activation output.

$$z^{(l)} = W^{(l)}x + b^{(l)} \tag{3.21}$$

Batch normalization is then applied to normalize $z^{(l)}$ as in eq. (3.22) where $\mu^{(l)}$ and $\sigma^{(l)2}$ are the mean and variance of $z^{(l)}$, and ϵ is a small constant for numerical stability. The normalized output is scaled and shifted using learnable parameters $\gamma^{(l)}$ and $\beta^{(l)}$.

$$\hat{z}^{(l)} = \frac{z^{(l)} - \mu^{(l)}}{\sqrt{\sigma^{(l)2} + \epsilon}} \tag{3.22}$$

$$z'^{(l)} = \gamma^{(l)}\hat{z}^{(l)} + \beta^{(l)} \tag{3.23}$$

Finally, a ReLU activation function is applied following eq. (3.24) where $h^{(l)}$ is the output of the layer. This process is repeated for each layer in the MLP except for the last layer, in which only the MLP layer is applied.

$$h^{(l)} = \text{ReLU}(z'^{(l)}) = \max(0, z'^{(l)}) \tag{3.24}$$

During the graph construction phase, a heterogeneous graph is created where the nodes represent tasks and VMs, with their respective embeddings serving as the node features (Figure 3.6). The edges in the graph define the relationships between these nodes. For task-task edges, the adjacency matrix $A_{J_i,J_j,t}$ is used to derive the dependencies in the DAG, connecting tasks based on their precedence constraints. Edges between task nodes and VM nodes represent the assignability of tasks to VMs and are derived from the compatibility matrix $I^{\text{cmp}}_{J_i,\text{VM}_j,t}$. Thus, an edge (J_i, VM_j) exists only if task J_i is compatible with VM_j. There are no edges inter-connecting VM nodes directly.

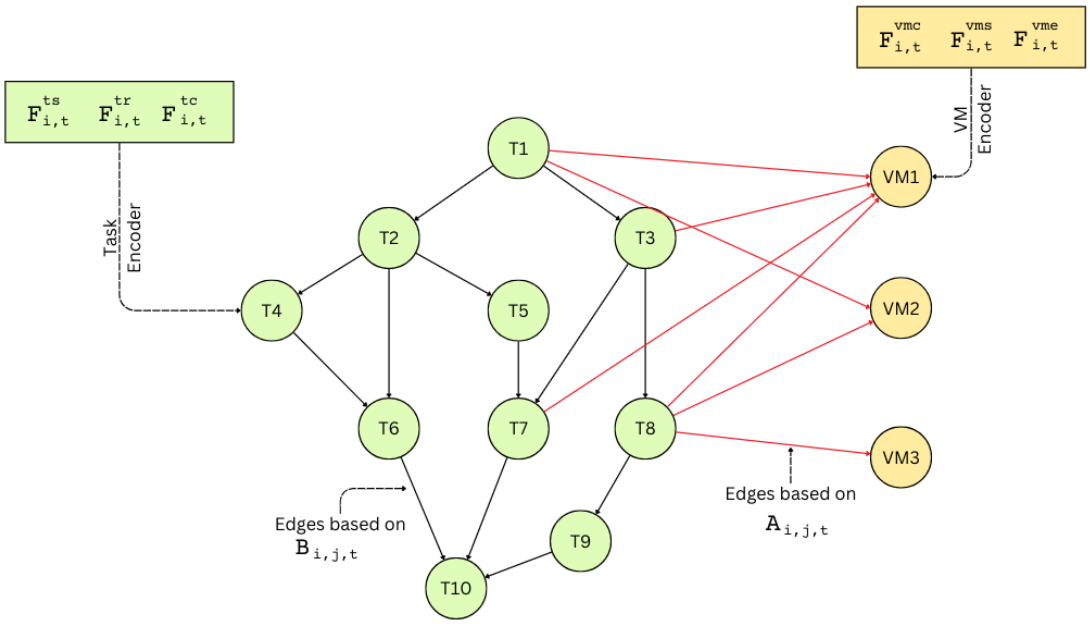

Fig. 3.6: An example of a graph constructed from a workflow

The constructed graph is processed through several GIN layers [26], which update the node embeddings by aggregating information from neighboring nodes. Let $h_v^{(k)}$ denote the embedding of node v at the k-th layer, then $h_v^{(k)}$ can be derived from the equation shown in eq. (3.25) where $\mathcal{N}(v)$ is the set of neighbors of node v, and $\epsilon^{(k)}$ is a learnable parameter.

$$h_v^{(k)} = \text{MLP}^{(k)}\left((1+\epsilon^{(k)}) \cdot h_v^{(k-1)} + \sum_{u \in \mathcal{N}(v)} h_u^{(k-1)}\right) \tag{3.25}$$

The overall graph embedding h_G is obtained using mean pooling over all node embeddings in the last layer L as in eq. (3.26) where N is the number of nodes in the graph $(N_J + N_{\text{VM}})$.

$$h_G = \frac{1}{N} \sum_{i=1}^{N} h_{v_i}^{(L)} \tag{3.26}$$

After the GIN layers, the updated task node embeddings h_{J_i} and VM node embeddings h_{VM_j} are extracted from the embedding of the last layer L. Then, for all compatible task-VM pairs, the action embeddings are created by concatenating the corresponding task node embedding, VM node embedding, and the graph embedding by following eq. (3.27).

$$h_{J_i,\text{VM}_j} = h_{J_i} \,||\, h_{\text{VM}_j} \,||\, h_G \tag{3.27}$$

These action embeddings are passed through the graph scorer, an MLP similar to the encoders but without batch normalization, to produce a scalar score s_{J_i,VM_j} for each action.

An action matrix $S \in \mathbb{R}^{N \times M}$ is formed, where invalid actions (incompatible task-VM pairs) are assigned $-\infty$ to mask them, and valid actions are assigned their corresponding scores. The policy is then obtained by applying the softmax function over the action matrix as in eq.(3.28) where $\mathcal{A}$ is the set of valid actions.

$$\pi(J_i, \text{VM}_j) = \frac{\exp(s_{J_i,\text{VM}_j})}{\sum_{k,l \in \mathcal{A}} \exp(s_{J_k,\text{VM}_l})} \tag{3.28}$$

The above architecture is the architecture of the actor network. The actor network is responsible for generating a policy by determining the probabilities of assigning tasks to VMs. The critic network, on the other hand, is responsible for evaluating the current state by estimating the expected cumulative reward. It shares the same architecture as the actor network up to the GIN layers. The graph embedding h_G, obtained after mean pooling from the GIN layers, is passed through a value scorer, an MLP without batch normalization, to produce a scalar value estimation. This value represents the expected cumulative reward from the state s_t.

3.2.6 Dataset Generation

Each workflow is represented as a DAG generated using the $G(n, p)$ method, where n is the number of nodes and p is the probability of creating an edge between any two nodes. The probability p is defined as $\frac{\log(n+\epsilon)}{n}$, where ϵ is a small constant to ensure a high probability of avoiding isolated vertices [133]. The number of nodes n in each generated DAG is set to $N^J - 1$, and a start node is added to ensure the DAG remains connected.

The computational workload of each task, represented as its length, is constrained to the range $[L^{\min}, L^{\max}]$. Task lengths are sampled from a normal distribution with a mean of $\frac{L^{\min}+L^{\max}}{2}$ and a standard deviation of $\frac{L^{\max}-L^{\min}}{6}$, ensuring most values fall

within the specified range. Additionally, datasets with left-skewed and right-skewed distributions were generated to explore different task distributions and their impact on scheduling. These skewed distributions were created using skew-normal distributions, where left-skewed datasets prioritize shorter tasks and right-skewed datasets favor longer tasks, with the skewness (α) controlled around the same mean and standard deviation as the normal distribution. For this study, $\alpha = -5$ was used to generate left-skewed distributions, while $\alpha = 5$ was used to generate right-skewed distributions, ensuring a substantial skewness. This variety in datasets provides a comprehensive evaluation of the scheduling method under diverse task workload scenarios.

For each dataset, the number of VMs is set to N^{VM}, and the number of hosts to N^{H}. VMs are assigned to hosts randomly. Each VM's allocated memory is chosen randomly between R^{min} and R^{max}, while the memory requirements of tasks are capped to ensure compatibility with VM capacities. The CPU speeds for each VM is randomly set between $S_{\text{VM}}^{\text{min}}$ and $S_{\text{VM}}^{\text{max}}$. Host characteristics, such as CPU speed and power consumption, are derived from the SPECpower benchmark suite [47] and listed in Table 3.2.

TABLE 3.2: HOST CHARACTERISTICS USED FOR THE DATASET GENERATION

Name	**Number of Cores**	**CPU Speed (GIPS)**	**Idle Power (W)**	**Peak Power (W)**
PowerEdge R740	56	604.8	50	432
PowerEdge C6100	48	587.52	227	895
PowerEdge R820	32	332.8	110	446
iDataPlex dx360	16	187.712	116	475
PowerEdge R710	12	146.88	62	227

Note. Dataset taken from SPECpower_ssj® 2008 benchmark suite

The number of workflows in each dataset is randomly chosen between 1 and N^{W}, creating varied levels of task concurrency.

3.2.7 Agent Training

For training the RL agent, a dataset was generated with the parameters shown in Table 3.3.

The training of the reinforcement learning agent was carried out using the PPO algorithm. The implementation was done in Python following the CleanRL implementation [134]. The training was conducted over a fixed number of episodes, with each episode representing a complete scheduling scenario. At the end of each episode, the agent's policy was updated using PPO to balance exploration and exploitation. The hyperparameters used for training the agent are summarized in Table 3.4.

TABLE 3.3: TRAINING DATASET GENERATION PARAMETERS

Parameter	Description	Value
N^H	Number of hosts	4
N^{VM}	Number of VMs	10
N^W	Number of workflows	10
N^J	Number of tasks in a workflow	20
S_{VM}	CPU speed of a VM in MIPS	[500, 5000]
R	Amount of RAM for a VM (in GB)	[1, 10]
L	Task length	[500, 100000]

TABLE 3.4: HYPERPARAMETERS USED FOR PPO TRAINING

Parameter	Description	Value
lr	Learning rate	2.5×10^{-4}
N	Batch size	512
γ	Discount factor	0.99
λ	Generalized Advantage Estimation (GAE) factor	0.95
ϵ	Clipping parameter	0.2
K	Number of epochs per PPO update	4
β	Entropy coefficient	0.01
c	Maximum gradient norm for clipping	0.5
B	Number of steps per policy rollout	128
M	Number of minibatches per update	4
θ_{je}	Job encoder hidden layers	$[32, 32]$
θ_{me}	Machine encoder hidden layers	$[32]$
θ_{ee}	Edge encoder hidden layers	$[32]$
θ_{ee}	Edge scorer hidden layers	$[64, 32]$
θ_{ee}	Graph scorer hidden layers	$[32, 32]$
E	Number of features in embeddings	32
L_G	Number of GIN layers	3
α	Weight of makespan reward	1
β	Weight of energy consumption reward	1

Figure 3.7 illustrates the learning curve of the agent, showing the increase in episodic return as training progressed. The training results indicate that the PPO algorithm, in conjunction with the GIN-based architecture, effectively learned to minimize the makespan and energy consumption while handling dynamic task arrivals and dependencies.

3.2.8 Cloud Computing Simulation Model

The evaluation of the proposed algorithm is conducted in a simulated cloud environment. Most research on cloud task and cloud workflow scheduling has utilized CloudSim [19], a widely used Java-based, open-source tool for simulating cloud environments [3, 16]. CloudSim provides a flexible and extensible framework specifically designed for

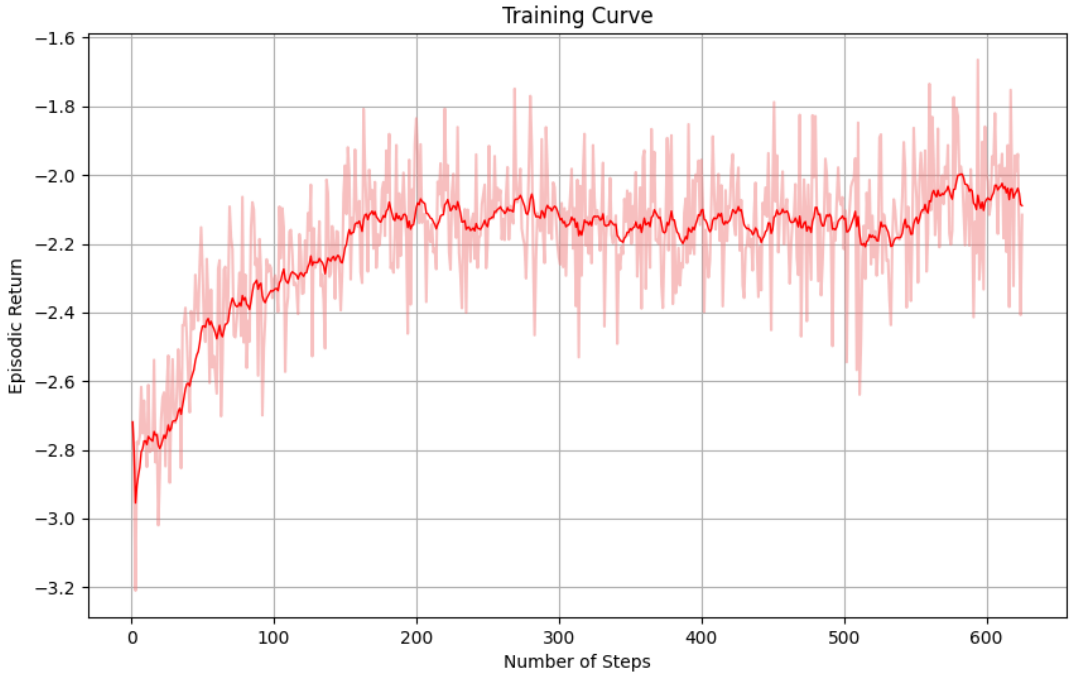

Fig. 3.7: Episodic return during training

modeling and simulating cloud data centers and resource management strategies.

The proposed simulation model represents a multi-layered cloud-computing environment designed to capture workflows, task dependencies, and resource dynamics. The architecture consists of multiple components, including datacenters, VMs, and cloudlets (tasks), where each workflow is represented as a DAG of tasks. The Cloud Information Service (CIS) maintains a registry of all available resources across datacenters, including VM configurations and their current states.

The Datacenter Broker acts as the intermediary between the user-submitted workflows and the underlying resources. The broker uses the implemented scheduler to dynamically allocate tasks to VMs. As depicted in Figure 3.8, tasks within workflows are dynamically assigned to VMs allocated across multiple hosts. This setup effectively emulates real-world scenarios where tasks compete for resources in heterogeneous environments.

The simulation supports various configurations, including the number of hosts and VMs. Workflow tasks are characterized by computational requirements (measured in Million Instructions) and memory needs, while VMs are configured with specific CPU speeds, RAM allocations, and energy consumption models. These configurations are based on parameters derived from real-world datasets and the SPECpower benchmark [47], ensuring heterogeneity in the simulation environment.

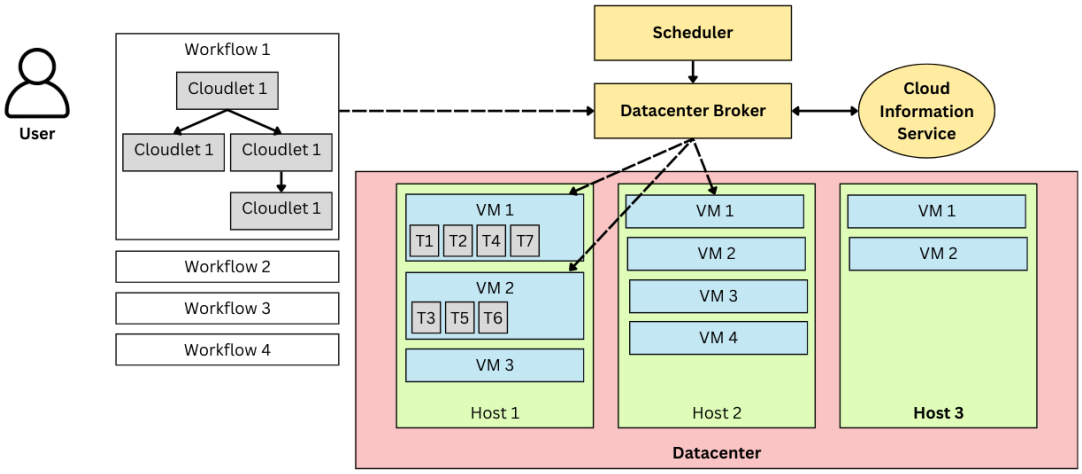

Fig. 3.8: Cloud-computing infrastructure and components assumed in the model architecture used for simulation developed in CloudSim

3.2.9 Agent Evaluation

For evaluating the RL agent, a dataset was generated with attributes given in table 3.5. During evaluation, the number of workflows and tasks is varied to assess the model's performance across several scenarios.

TABLE 3.5: EVALUATION DATASET GENERATION PARAMETERS

Parameter	**DS**$_1$	**DS**$_2$	**DS**$_3$
Number of hosts (N^H)	3	3	3
Number of VMs (N^{VM})	4	5	5
Number of workflows (N^W)	10	10	10
Number of tasks in a workflow (N^J)	$[1, 5]$	$[15, 20]$	$[40, 50]$
CPU speed of a VM in MIPS (S_{VM})	$[2500, 5000]$	$[1000, 5000]$	$[1000, 5000]$
Amount of RAM for a VM (R)	$[1, 2]$	$[1, 10]$	$[1, 10]$
Task length (L)	$[50000, 100000]$	$[500, 100000]$	$[500, 1000000]$

The current configuration of the CloudSim environment provides a controlled and manageable setup for evaluating the proposed algorithm under a variety of conditions. This configuration is designed to balance computational feasibility with the ability to assess the algorithm's performance across diverse scenarios. By focusing on small to medium-scale environments, the evaluation ensures clear insights into the algorithm's capabilities, particularly for foundational testing and validation. For instance, in DS_3 tasks in the dataset are generated with lengths ranging from 500 to 100,000 MI, sampled from a normal distribution with a mean and a standard deviation calculated as described in section 3.2.6. Workflows are modeled as DAGs of 40-50 tasks per workflow, and task dependencies are determined using an edge probability of $\frac{\log(n+\epsilon)}{n}$ where ϵ is 0.1. The simulation includes a datacenter with 3 hosts where each host is defined

using specifications from SPEC benchmark. Up to 5 VMs are hosted with each VM having a maximum capacity of 10 GB RAM. For a simulation environment, this dataset represents a moderately scaled cloud infrastructure, providing a balanced mix of complexity and manageability.

Additionally, two additional datasets, DS_3^L and DS_3^R, were generated as left and right-skewed variants of DS_3, respectively. These datasets were created using the same configuration parameters described in Section 3.2.6, with the task length distribution adjusted to introduce skewness. The left-skewed dataset (DS_3^L) emphasizes shorter task lengths, while the right-skewed dataset (DS_3^R) focuses on longer task lengths, providing insights on method's performance on different workload conditions.

The parameter values in Table 3.5 were selected to simulate diverse scenarios representing small to medium-sized cloud environments. A modest number of hosts and VMs were chosen to reflect realistic configurations often encountered in private or hybrid cloud setups, balancing computational capacity with practical constraints. The variation in the number of tasks and their dependencies (small, medium, and large DAGs) was introduced to capture diverse workflow complexities, ranging from simple workflows with few tasks to complex workflows with extensive interdependencies. Task characteristics, such as length, were varied across datasets to assess the scheduler's adaptability to workloads with differing computational demands. Additionally, skewed datasets were generated by applying left and right skewness to the task lengths. This was done to introduce variability and test the robustness of the proposed model under scenarios where task distributions deviate from normal patterns, as might occur in dynamic real-world environments.

TABLE 3.6: SUMMARY OF COMPARISON ALGORITHMS

Algorithm	**Description**
Random	Randomly assigns tasks to VMs.
Round Robin [135]	Cyclically assigns tasks to VMs.
Min-Min [135]	Assigns tasks with the shortest execution time to the VM that completes them the fastest.
Max-Min [135]	Assigns tasks with the longest execution time to the VM with the earliest completion time.
HEFT [135]	Schedules workflows individually using the Heterogeneous Earliest Finish Time algorithm.

The proposed algorithm is compared against several baseline algorithms to comprehensively evaluate its performance and highlight its advantages. Table 3.6 summarizes the methods used, including HEFT, Min-Min, Max-Min, Round-Robin, and Random. These algorithms were chosen not only for their established roles in task scheduling but also to provide a diverse set of perspectives on scheduling performance. HEFT, Min-Min, and Max-Min represent widely recognized heuristic strategies that excel in specific

static or single-objective scenarios, making them valuable baselines for understanding fundamental scheduling behaviors. Round-Robin and Random scheduling, while simplistic, serve as essential comparisons to underscore the significance of structured decision-making and dynamic adaptability. Though these algorithms may not match the sophistication of GNN-Flat in terms of dynamic adaptability or multi-objective optimization, their inclusion highlights the limitations of static and heuristic-based approaches, emphasizing the superior adaptability and versatility of GNN-Flat. By comparing against algorithms with varying levels of complexity, the study ensures a comprehensive evaluation, demonstrating how GNN-Flat consistently outperforms both rudimentary and heuristic-based methods across diverse scheduling scenarios.

Furthermore, this comparison aims to validate the effectiveness of GNN-based models in cloud workflow scheduling, demonstrating that a learning-based approach can successfully model task dependencies and dynamic resource allocations while achieving competitive scheduling performance. By illustrating how GNN-Flat outperforms traditional methods, this study establishes GNN-based reinforcement learning as a promising direction for future research in cloud workflow scheduling.

3.3 Multi-Objective Cloud Workflow Scheduling with Two-Stage Deep Reinforcement Learning

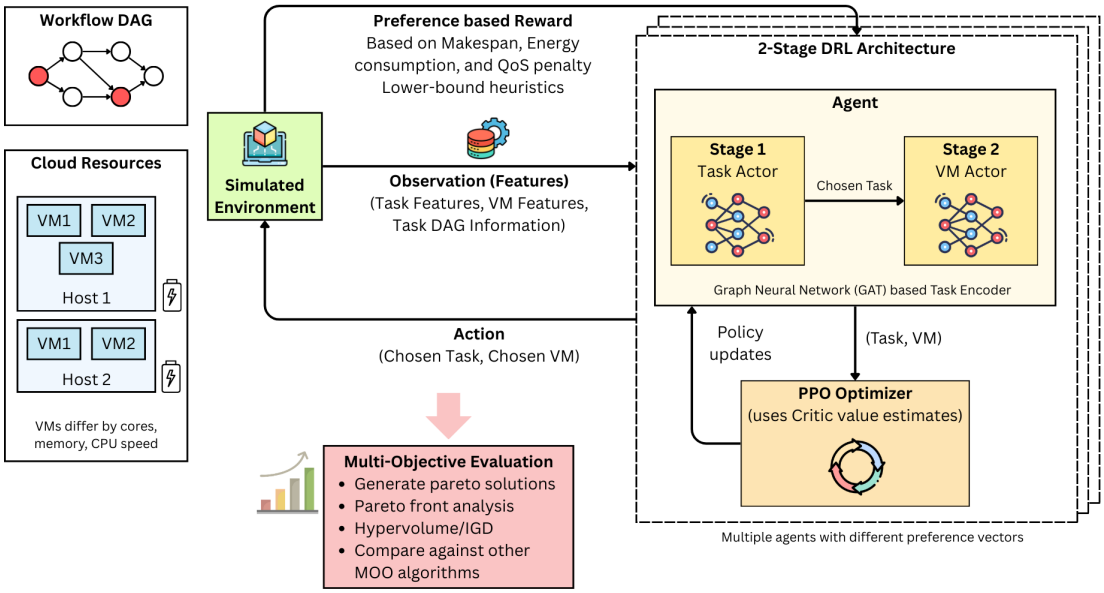

Fig. 3.9: Overview of the 2SD-GAT reinforcement learning methodology

Figure 3.9 presents the architectural overview of the DRL methodology adopted in 2SD-GAT study. The process begins with the generation of synthetic datasets under various parametric configurations. A simulated cloud environment provides the foundation for training, where the DRL agent interacts with the environment by

performing scheduling actions and receiving feedback in the form of rewards, updated states, and environmental observations. Through iterative learning, the agent aims to optimize multiple objectives: minimizing makespan, reducing energy consumption, and lowering QoS-related penalties, the latter modeled as the cumulative latency of high-priority tasks.

The subsequent sections are structured as follows. The formal model formulation that underpins both dataset generation and the simulated environment is first presented. Lower-bound heuristics, which play a crucial role in reward formulation, are then introduced. This is followed by a discussion of the data preprocessing and feature extraction process. The reinforcement learning framework is then detailed, followed by the preference-based reward function designed for multi-objective optimization. Finally, the agent architecture and model design are elaborated, with particular emphasis on the integration of GNNs, specifically the GAT.

3.3.1 Problem Modeling

In this work, workflows continue to be represented as DAGs, encoding task execution precedence, consistent with the modeling approach in Section 3.2.1. The overall workflow completion time, or makespan M, and the cumulative data center energy consumption E^{total} are retained as key optimization targets, defined respectively in eq. (3.1) and eq. (3.4).

Unlike the previous formulation, this model enhances the resource compatibility constraint by incorporating both memory and CPU core requirements. A task J_i is eligible to be assigned to virtual machine VM_j only if its memory and core demands are fully met, i.e., $R_{J_i}^{\text{mem}} \leq R_{\text{VM}_j}^{\text{mem}}$ and $R_{J_i}^{\text{core}} \leq R_{\text{VM}_j}^{\text{core}}$. Here, $R_{J_i}^{\text{mem}}$ and $R_{J_i}^{\text{core}}$ denote the memory and CPU core demands of task J_i, while $R_{\text{VM}_j}^{\text{mem}}$ and $R_{\text{VM}_j}^{\text{core}}$ refer to the respective capacities of VM j. Upon satisfying these conditions, the task execution time T_{J_i,VM_j} is computed using eq. (3.3).

An additional performance criterion introduced in this study is QoS sensitivity, captured through a latency penalty model. For simplicity, two tiers of tasks were considered: low-priority and high-priority. The scheduler is expected to prioritize high-priority tasks during scheduling decisions. To quantitatively capture this behavior, we define the QoS penalty score (Q) as shown in eq. (3.29).

$$Q = \sum_{J_i \in V} Q_{J_i} \tag{3.29}$$

Here, Q_{J_i} represents the QoS penalty score for task J_i, defined in eq. (3.30).

$$Q_{J_i} = t_{J_i}^{\text{start}} \cdot R_{J_i}^{\text{priority}} \tag{3.30}$$

In this formulation, $t_{J_i}^{\text{start}}$ denotes the start time of task J_i, and $R_{J_i}^{\text{priority}} \in \{0, 1\}$ is a

binary priority weight indicating whether the task is high-priority (1) or low-priority (0). This priority is assigned at the workflow level, meaning all tasks within a given workflow inherit the same priority as their parent workflow.

The objective is to find an optimal scheduling policy that minimizes makespan (M), total energy consumption (E^{total}), and QoS latency (Q), as formulated in eq. (3.31), subject to the specified constraints (addressing **RQ1**).

$$\text{minimize } \left(M, E^{\text{total}}, Q\right) \tag{3.31}$$

This optimization is subject to the constraints: (1) Tasks must satisfy task precedence constraints defined by the DAG, (2) Tasks must satisfy memory requirements, $R^{\text{mem}}_{J_i} \leq R^{\text{mem}}_{\text{VM}_j}$, and (3) Tasks must satisfy core count requirements, $R^{\text{core}}_{J_i} \leq R^{\text{core}}_{\text{VM}_j}$.

3.3.2 Lower Bound Heuristics

As with the approach detailed in Section 3.2.1, this study employs lower-bound estimators to derive optimistic reference values for makespan, energy consumption, and QoS penalties (addressing **RQ3**). These lower bounds play a key role in shaping the agent's reward structure by providing theoretically minimal values for each objective. The makespan lower-bound at decision step t, denoted as M^{est}_t, follows the formulation in eq. (3.10). Similarly, the estimated minimum active energy consumption, E^{est}_t, is given by eq. (3.13).

This study further introduces a heuristic estimate for the lowest possible QoS penalty score at time t, accounting for scheduling priority. It is computed by estimating the earliest possible start time for each task, weighted by its priority. For scheduled tasks, the actual start time is used directly. For unscheduled tasks, the earliest feasible start time is determined by evaluating all compatible VMs and accounting for both parent task completion and VM readiness. This estimation provides an optimistic lower bound on the cumulative QoS penalty score under ideal scheduling conditions.

Formally, the QoS penalty score lower-bound estimation, denoted as Q^{est}_t, is defined in eq. (3.32).

$$Q^{\text{est}}_t = \sum_{J_i \in V} Q^{\text{est}}_{J_i,t} \tag{3.32}$$

Here, V denotes the set of all tasks, and $Q^{\text{est}}_{J_i,t}$ represents the estimated QoS penalty score of task J_i at time t, defined in eq. (3.33).

$$Q^{\text{est}}_{J_i,t} = \begin{cases} 0 & \text{if } R^{\text{priority}}_{J_i} = 0, \\ t^{\text{start}}_{J_i} & \text{if } J_i \text{ is scheduled}, \\ \min\limits_{\text{VM}_j \in \text{VM}^{\text{cmp}}_{J_i}} \left(\max\limits_{J_k \in J^{\text{par}}_{J_i}} \left(M^{\text{est}}_{J_k,t}, M_{\text{VM}_j,t} \right) \right) & \text{otherwise.} \end{cases} \tag{3.33}$$

As in the makespan estimation, $\text{VM}^{\text{cmp}}_{J_i}$ is the set of compatible VMs for task J_i, $J^{\text{par}}_{J_i}$ denotes its parent tasks, and $M_{\text{VM}_j,t}$ is the current finish time of VM_j as defined in eq. (3.12).

It is important to note that these lower bounds are used for reward normalization and shaping rather than as optimization targets themselves. Consequently, the learning process depends primarily on relative performance differences between actions, not on the absolute tightness of the bounds. Even when the estimates are loose or imperfect, they still provide a consistent optimistic reference scale that stabilizes reward magnitudes across workflows of different sizes and characteristics. In practice, this makes the DRL policy robust to moderate estimation errors, as suboptimal bounds may affect normalization scale but do not alter the underlying environment transitions or true objective evaluations.

3.3.3 Data Preprocessing

To facilitate efficient training and accelerate convergence, the raw simulation data is transformed into a structured representation for the 2SD-GAT pipeline. This preprocessing phase extracts and organizes information at the task and VM levels, while also encoding the task precedence relationships as a graph structure. A summary of the extracted input features is provided in Table 3.7.

3.3.4 Reinforcement Learning Framework

The RL paradigm adopted in this section extends the principles introduced in Section 3.2.4, with a particular emphasis on structuring the decision-making process to address the increased complexity of multi-objective cloud workflow scheduling.

In complex scheduling environments such as cloud task scheduling, two-stage reinforcement learning can be used to decompose decision-making into multiple levels. A common two-stage architecture involves agents operating in sequence: the task agent selects the next task to schedule, while the VM agent selects the most suitable VM for executing the chosen task. This decoupled structure simplifies the action space, reduces combinatorial complexity, and allows for specialized learning at each level. Similar two-stage approaches have also been applied to the FJSSP, where one agent selects the

TABLE 3.7: SUMMARY OF INPUT FEATURES

Feature	Symbol	Name	Description
$F_{i,t}^{1}$	$I_{J_i,t}^{\text{ready}}$	Task schedulable	Binary flag indicating if task J_i is ready to be scheduled (i.e., all parent tasks are completed and J_i has not yet been scheduled).
$F_{i,t}^{2}$	$I_{J_i,t}^{\text{sch}}$	Task scheduled	Binary flag indicating whether task J_i is already scheduled.
$F_{i,t}^{3}$	$M_{J_i,t}^{\text{est}}$	Task completion time	Estimated completion time of task J_i if unscheduled, or actual completion time if already scheduled. (Equation 3.11)
$F_{i,t}^{4}$	$E_{J_i,t}^{\text{est}}$	Task energy consumption	Estimated or actual energy consumption of task J_i based on its assignment. (Equation 3.14)
$F_{i,t}^{5}$	$Q_{J_i,t}^{\text{est}}$	Task QoS penalty score	Estimated or actual QoS penalty score for task J_i, weighted by its priority. (Equation 3.33)
$F_{i,t}^{6}$	$R_{J_i}^{\text{priority}}$	Task priority	The QoS-priority level of task J_i (e.g., 0 or 1).
$F_{j,t}^{7}$	$M_{\text{VM}_j,t}$	VM completion time	Estimated time when VM j becomes available to execute a new task. (Equation 3.12)
$F_{i,j,t}^{8}$	$I_{J_i,\text{VM}_j,t}^{\text{cmp}}$	VM compatibility	Binary flag indicating whether VM j is compatible with task J_i, based on resource constraints.
$F_{i,j,t}^{9}$	$T_{J_i,\text{VM}_j,t}$	Execution time cost	Execution time required for task J_i on VM j, considering VM specs and task length.
$F_{i,j,t}^{10}$	$E_{J_i,\text{VM}_j,t}$	Energy consumption cost	Estimated energy consumed by task J_i if executed on VM j.
$A_{i,j,t}$	$I_{J_i,J_j,t}^{\text{dep}}$	Task dependency flag	Binary flag representing whether there is a dependency (directed edge) from task J_i to task J_j. If a task is scheduled to a VM, a new dependency is added between the previous task of the VM and the scheduled task.

next operation and another determines the machine to execute it, effectively handling task-machine assignment under complex constraints [56].

In this research, PPO is applied to the cloud workflow scheduling problem, where the agent's goal is to minimize several objectives. The reward R_t is designed to reflect the multi-objective optimization framework's focus, penalizing the agent for longer makespans, higher energy usage, and delayed tasks. To normalize the these

reward segments, a normalization method is employed. The reward function is given in eq. (3.34) where M_t^{est}, E_t^{est}, and Q_t^{est} are the makespan lower-bound, active energy consumption lower-bound, and QoS penalty score lower-bound estimations for the state S_t (eq. (3.10), eq. (3.13), and eq. (3.32)). Parameters α, β, and γ balance the importance of minimizing makespan and energy consumption.

$$R_{t+1} = -\alpha \left(\frac{M_{t+1}^{\text{est}} - M_t^{\text{est}}}{M_{t+1}^{\text{est}}} \right) - \beta \left(\frac{E_{t+1}^{\text{est}} - E_t^{\text{est}}}{E_{t+1}^{\text{est}}} \right) - \gamma \left(\frac{Q_{t+1}^{\text{est}} - Q_t^{\text{est}}}{Q_{t+1}^{\text{est}}} \right) \tag{3.34}$$

The normalization factors α, β, and γ ensure that each objective contributes on a comparable scale, preventing any single objective from dominating the reward signal due to differences in magnitude. These factors are determined empirically using the method described in Section 3.3.9.

It is important to note again that the reward signal is computed from the change in the agent's estimated objective values before and after an action is taken. As shown in eq. (3.34), the reward depends on the relative difference between the estimated makespan, energy, and QoS latency at time t and their updated estimates at time $t + 1$. This formulation measures how much the agent's decision improves or worsens the expected outcomes, rather than relying on absolute objective magnitudes. Because the estimator is intentionally optimistic (actual values will always exceed these estimates), the agent is penalized whenever its action leads to a deterioration in the predicted objectives. This encourages consistently improving decisions and provides a more stable learning signal by normalizing objective changes and reducing variance across workflows of different scales.

3.3.5 Preference-Based Reward Function

To generate a Pareto front in multi-objective optimization, it is essential to obtain a diverse set of solutions that reflect different trade-offs between competing objectives. In this study, this is achieved by introducing a preference-based reward function. During training, each agent is assigned a set of preference weights representing its emphasis on specific objectives. This enables the same architecture to be trained under different objective priorities, producing a spectrum of solutions. When combined, these agents contribute to the construction of a Pareto front by spanning different regions of the objective space.

The reward function is defined as a weighted combination of changes in estimated makespan, energy consumption, and QoS penalty score, as shown in eq. (3.35). The preference weights for each objective are denoted by λ_1, λ_2, and λ_3, and are fixed per agent during training. Mathematically, the scalar reward R_{t+1} is a linear combination of three normalized objective changes, weighted by the preference vector. This formulation

allows the agent to learn a policy that optimizes a specific weighted combination of objectives, and by training multiple agents with different preference vectors, a diverse set of Pareto-optimal policies can be obtained.

$$R_{t+1} = -\lambda_1 \alpha \left(\frac{M^{\text{est}}_{t+1} - M^{\text{est}}_{t}}{M^{\text{est}}_{t+1}} \right) - \lambda_2 \beta \left(\frac{E^{\text{est}}_{t+1} - E^{\text{est}}_{t}}{E^{\text{est}}_{t+1}} \right) - \lambda_3 \gamma \left(\frac{Q^{\text{est}}_{t+1} - Q^{\text{est}}_{t}}{Q^{\text{est}}_{t+1}} \right) \quad (3.35)$$

The preference weights λ_1, λ_2, and λ_3 are systematically enumerated using a uniform grid of the objective–weight space. Let $\boldsymbol{\lambda}^{(k)} = (\lambda_1^{(k)}, \lambda_2^{(k)}, \lambda_3^{(k)})$ denote the preference vector assigned to agent k. Each weight $\lambda_i^{(k)}$ is drawn from a discrete, uniformly spaced set over the interval $[0,\ 1]$, defined as eq. (3.36) where Δ denotes the grid resolution. Although alternative sampling strategies (e.g., simplex sampling or adaptive sampling) could have been used, the grid–search method was chosen for its simplicity and for providing explicit coverage of the objective–weight space.

$$\Lambda = \{\lambda \mid \lambda = n\Delta,\ n \in \mathbb{N},\ 0 \leq \lambda \leq 1\} \quad (3.36)$$

The complete set of candidate preference vectors is then constructed as the Cartesian product as in eq. (3.37).

$$\mathcal{P} = \Lambda \times \Lambda \times \Lambda \quad (3.37)$$

To avoid redundant objective prioritizations, preference vectors that differ only by a positive scalar factor are removed, as they induce equivalent reward scaling and thus identical optimization behavior. Two preference vectors $\boldsymbol{\lambda}^{(i)}$ and $\boldsymbol{\lambda}^{(j)}$ are considered redundant if $\boldsymbol{\lambda}^{(i)} = c\,\boldsymbol{\lambda}^{(j)}, \qquad c > 0.$

One limitation of the fixed preference approach is that it requires training multiple agents to cover the Pareto front comprehensively. An alternative approach would be to use a single agent with dynamic preference conditioning, where the preference vector is provided as an additional input to the policy network. This would allow a single agent to generate solutions for arbitrary preference vectors at inference time. However, this approach was not pursued in this study due to the increased complexity of the training procedure and the potential for reduced solution quality compared to dedicated agents.

The diversity and coverage of the Pareto front are evaluated using standard multi-objective metrics such as Hypervolume and Inverted Generational Distance, as discussed in the Section 4. These metrics provide quantitative evidence that the preference-based training strategy successfully generates a well-distributed set of Pareto-optimal solutions.

3.3.6 Agent Architecture and Model Design

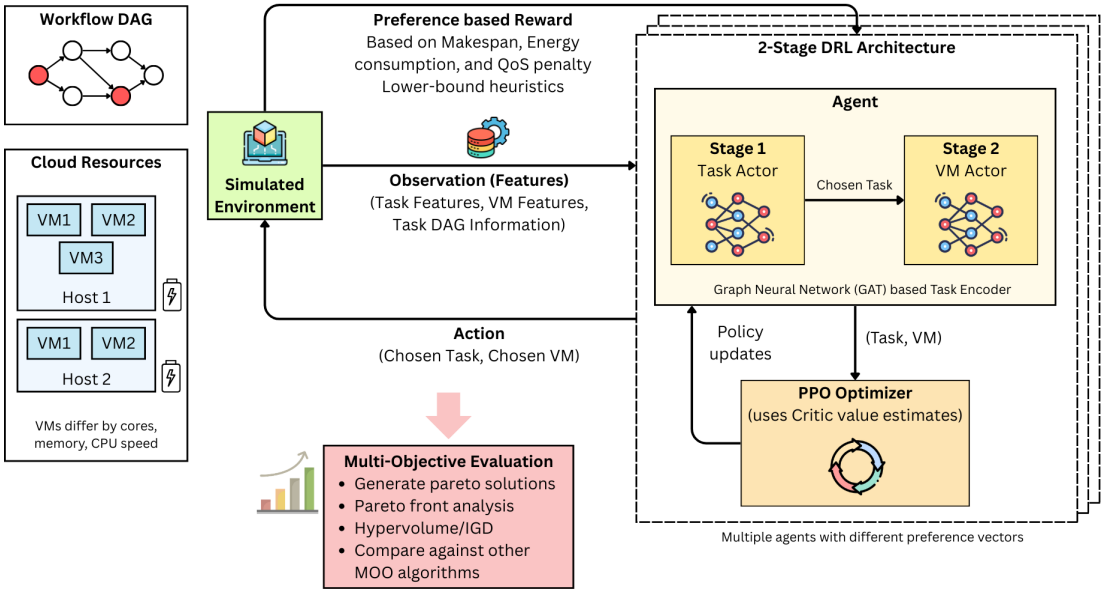

Fig. 3.10: RL Framework used for 2SD-GAT

To provide additional intuition into the agent's decision-making, the scheduling process can be viewed as a dynamic project management problem in which the workflow DAG evolves as tasks are scheduled and completed. At each decision step, the agent observes updated task states (e.g., ready and scheduled indicators), VM states, and the current dependency structure. These are encoded by the GAT to produce task embeddings that reflect both local properties and global structural roles, such as downstream depth and dependency influence. For instance, in a simple workflow where a preprocessing task must complete before two successor tasks can start, only the preprocessing task is initially marked ready. Once it finishes, the successor tasks become ready and the graph state is updated. The GAT then recomputes embeddings on the current DAG, allowing the policy to reassess which task has greater impact on future makespan, energy, and QoS objectives. The Task Actor selects a task accordingly, and the VM Actor assigns it to a suitable resource based on current VM conditions and objective trade-offs.

Unlike fixed heuristics, the learned policy is not a rigid rule-based procedure. It is optimized for long-term cumulative reward and may therefore choose actions that are not immediately optimal but lead to better overall scheduling performance over time.

Figure 3.10 illustrates the overall data flow in the training architecture of 2SD-GAT. The environment exposes observations, including task features, VM features, and the workflow dependency graph, which are fed simultaneously to the Task Actor, VM Actor, and Critic. The Task Actor first selects the next task to schedule, which is then passed to the VM Actor to determine the most suitable virtual machine. The combined

action, consisting of the selected task and VM, is applied to the environment, which in turn returns the next observation and a scalar reward signal. The Critic receives both the environment's observation and the corresponding reward, and computes value estimates used to derive advantage signals. These advantage estimates, along with the policy outputs from both actors, are passed to the PPO optimizer. PPO then updates the parameters of the Critic and both Actors [132].

3.3.6.1 Stage 1: Task Actor Network

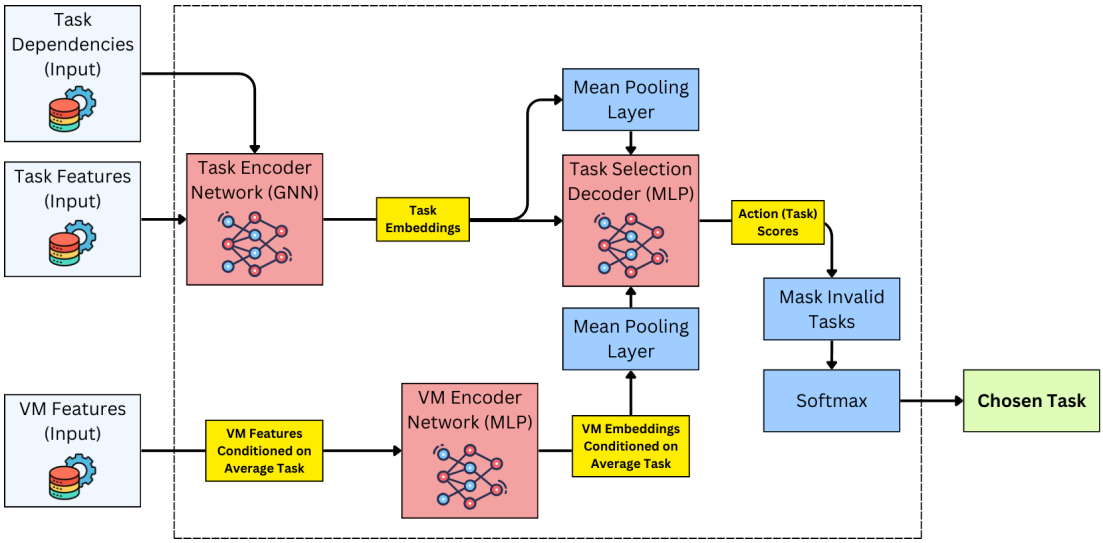

Fig. 3.11: Task Actor Network Architecture

The Task Actor is responsible for selecting the next task J_i to be scheduled from the workflow DAG at each time step. Its input includes the task feature matrix $X^{\text{task}} \in \mathbb{R}^{N_t \times F_t}$, the VM feature tensor $X^{\text{vm}} \in \mathbb{R}^{N_t \times N_v \times F_v}$, and the task dependency graph $A \in \{0,1\}^{N_t \times N_t}$, where N_t is the number of tasks, N_v is the number of VMs, and F_t, F_v are the dimensions of the task and VM features, respectively.

A core subcomponent of the Task Actor is the Graph Neural Network, implemented using a multi-layer Graph Attention Network. This GNN processes the workflow DAG to generate task embeddings that capture both local task properties and global structural dependencies. The same GAT-based task encoder is later reused by the VM Actor through the pooled task context p^{task}.

Each task J_i is represented as a node with an input feature vector $x_i \in \mathbb{R}^{F_t}$, formed by concatenating the task-level features $F^1_{i,t}$–$F^6_{i,t}$ listed in Table 3.7. These features encode the task's scheduling state and its estimated influence on the optimization objectives, providing the GNN with a rich descriptive basis for representation learning.

Task dependencies are expressed as directed edges in the workflow DAG. An edge (J_i, J_j) indicates that J_i must complete before J_j begins. These dependencies are stored

in the adjacency matrix A, where $A_{i,j} = 1$ signifies a directed edge from J_i to J_j. Since assigning a task to a VM introduces a new precedence constraint with the previously executed task on that VM, the matrix is updated at each DRL time step t to incorporate both the original workflow edges and these dynamically introduced dependencies.

Although the edges do not carry explicit features beyond their binary connectivity, the GAT architecture leverages its attention mechanism to implicitly infer the importance of each dependency based on the connected node features. This is particularly beneficial in workflow scheduling, where some dependencies, such as those along or near the critical path, have disproportionately strong influence on makespan and overall performance. Unlike conventional GNNs using uniform aggregation weights, the GAT dynamically computes attention coefficients that determine how much information each neighbor contributes to a node's updated representation. This enables the model to emphasize structurally important relationships and to highlight tasks whose scheduling order is more critical.

The choice of GAT is further motivated by its strong generalization capabilities. Because attention weights are computed from both node features and learned embeddings, the encoder naturally adapts to workflow DAGs of varying depth, width, and structural complexity. The dynamic attention computation also evolves throughout training, allowing the GNN to refine how it integrates information from predecessors and successors as representations become more expressive. The effectiveness of this GAT-based encoder is validated through an ablation study presented in Section 4.2.2.

Moreover, the encoder's understanding of the evolving DAG is reinforced by key state-indicative inputs, particularly the task-ready ($F^1_{i,t}$) and task-scheduled ($F^2_{i,t}$) status features, together with the dependency edges ($A_{i,j,t}$) that are updated after every scheduling decision. These signals allow the GAT to continually reassess which nodes and precedence relations are most influential as tasks progress through different scheduling stages. At each DRL time step, the GAT operates on the current adjacency matrix $A_{i,j,t}$, thus, task embeddings are recomputed on the updated graph, ensuring that each decision is conditioned on the latest workflow topology. As a result, the network maintains an up-to-date structural view of the workflow, helping the agent preserve stable convergence even when newly introduced constraints modify the graph topology.

Let $x_i \in \mathbb{R}^{F_t}$ be the input feature vector of task J_i. In a GAT with L layers, the output embedding at layer l is computed by aggregating information from the neighbors $\mathcal{N}(i)$ using learned attention weights as defined in eq. (3.38) [27].

$$h_i^{(l)} = \sigma\left(\sum_{j \in \mathcal{N}(i)} \alpha_{ij}^{(l)} W^{(l)} h_j^{(l-1)}\right), \quad h_j^{(0)} = x_j \tag{3.38}$$

The attention coefficients $\alpha_{ij}^{(l)}$ are computed using a shared attention mechanism that applies a LeakyReLU activation over concatenated linear projections of neighboring

node embeddings [27].

After L layers, the final task embedding for each task J_i is denoted as $h_i^{\text{task}} = h_i^{(L)} \in \mathbb{R}^E$. These embeddings are aggregated via mean pooling to generate a global task-level context vector (task pool) $p^{\text{task}} \in \mathbb{R}^E$ using eq. (3.39). The mean pooling layer aggregates task embeddings by computing their element-wise average across all tasks. Given N task embeddings, the pooled representation is obtained by averaging along the task dimension. This produces a global context representation while treating all tasks with equal importance. We adopt mean pooling due to its simplicity and stable aggregation behavior.

$$p^{\text{task}} = \frac{1}{N_t} \sum_{i=1}^{N_t} h_i^{\text{task}} \tag{3.39}$$

To incorporate information about VM availability, the average VM features across all tasks are computed for each VM. Let $X_{i,j}^{\text{vm}} \in \mathbb{R}^{F_v}$ denote the feature vector of VM j with respect to task i. The average VM feature representation $x_j^{\text{vm, avg}} \in \mathbb{R}^{F_v}$ is computed as in eq. (3.40).

$$x_j^{\text{vm, avg}} = \frac{1}{N_t} \sum_{i=1}^{N_t} X_{i,j}^{\text{vm}}, \quad \forall j \in [1, N_v] \tag{3.40}$$

These features are passed through the VM encoder network, which is a MLP, to generate VM embeddings $h_j^{\text{vm, avg}} \in \mathbb{R}^E$. Each MLP layer consists of an affine transformation followed by a non-linear activation. The affine transformation at layer l is computed using eq. (3.41), and the activation output is computed using eq. (3.42).

$$z^{(l)} = W^{(l)} x + b^{(l)} \tag{3.41}$$

$$h^{(l)} = \text{ReLU}(z^{(l)}) \tag{3.42}$$

After passing through the MLP, the resulting VM embeddings are aggregated using mean pooling to form the VM-level context vector (VM pool) $p^{\text{vm, avg}} \in \mathbb{R}^E$ as shown in eq. (3.43).

$$p^{\text{vm, avg}} = \frac{1}{N_v} \sum_{j=1}^{N_v} h_j^{\text{vm, avg}} \tag{3.43}$$

For each task J_i, the task embedding h_i^{task} is concatenated with the global context vectors p^{task} and $p^{\text{vm, avg}}$ to form a combined representation e_i^{task} as shown in eq. (3.44).

$$e_i^{\text{task}} = [h_i^{\text{task}} \,\|\, p^{\text{task}} \,\|\, p^{\text{vm, avg}}] \tag{3.44}$$

This vector is passed through a decoder MLP which outputs a scalar score s_i^{task} for each task. These scores are then masked using a binary mask that identifies schedulable tasks [56]. The masked score $\tilde{s}_i^{\text{task}}$ is defined in eq. (3.45).

$$\tilde{s}_i^{\text{task}} = \begin{cases} s_i^{\text{task}} & \text{if } J_i \text{ is ready to be scheduled,} \\ -\infty & \text{otherwise.} \end{cases} \tag{3.45}$$

Finally, the task policy π^{task} is obtained using the softmax function over the masked scores as shown in eq. (3.46).

$$\pi_i^{\text{task}} = \frac{\exp(\tilde{s}_i^{\text{task}})}{\sum_{j=1}^{N_t} \exp(\tilde{s}_j^{\text{task}})} \tag{3.46}$$

The selected task J_i is then sampled from this distribution during training or chosen using the maximum probability during evaluation.

3.3.6.2 Stage 2: VM Actor Network

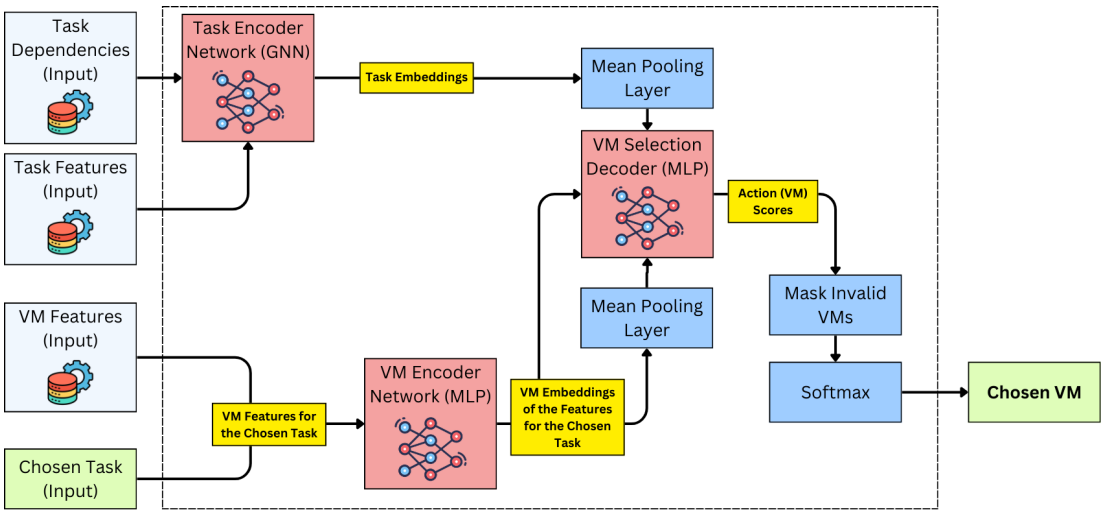

Fig. 3.12: VM Actor Network Architecture

The VM Actor is responsible for selecting an appropriate VM VM_j to execute the task J_i that was selected by the Task Actor. It uses the same VM encoder architecture (the MLP described in 3.3.6.1), but with a different choice of input features: since the task has already been chosen, the VM Actor only considers the corresponding VM feature slice $X_i^{\text{vm}} \in \mathbb{R}^{N_v \times F_v}$, which contains the VM features relevant to the selected task J_i.

These features are passed through the VM encoder network to generate VM embeddings $h_j^{\text{vm},i} \in \mathbb{R}^E$ for each VM j in the context of the chosen task i. The pooled

VM-level context vector $p^{\text{vm},i}$ is computed via mean pooling over these embeddings as in eq. (3.47).

$$p^{\text{vm},i} = \frac{1}{N_v} \sum_{j=1}^{N_v} h_j^{\text{vm},i} \tag{3.47}$$

To incorporate global context into the decision process, each VM embedding $h_j^{\text{vm},i}$ is concatenated with the task-level pooled vector p^{task} (computed by the GAT-based task encoder) and the VM-level pooled vector $p^{\text{vm},i}$ to form a combined representation e_j^{vm} for each VM as in eq. (3.48).

$$e_j^{\text{vm}} = \left[h_j^{\text{vm},i} \parallel p^{\text{task}} \parallel p^{\text{vm},i} \right] \tag{3.48}$$

These combined vectors are passed through a decoder MLP to produce a scalar score s_j^{vm} for each VM, which represents the likelihood of selecting that VM for task J_i. To enforce task-VM compatibility constraints, these scores are masked using a binary mask. The masked scores are defined in eq. (3.49).

$$\tilde{s}_j^{\text{vm}} = \begin{cases} s_j^{\text{vm}}, & \text{if VM is schedulable and compatible with } J_i, \\ -\infty, & \text{otherwise.} \end{cases} \tag{3.49}$$

The final VM selection policy, π_j^{vm}, is then computed by applying the softmax function over the masked scores similar to eq. (3.46). During training, the VM is sampled from this probability distribution, while during evaluation, the VM with the highest probability is selected.

3.3.6.3 Critic Network

The Critic network estimates the state value function $V(s)$, which represents the expected cumulative return starting from state s and following the current policy. This value estimate is used in PPO to compute the advantage function and guide policy updates.

The Critic receives the same observation as the actor, including task features, VM features, and the task dependency graph. The task features are encoded using the same GAT-based encoder as in the Task Actor to produce a pooled task representation $\mathbf{p}^{\text{task}}$. Similarly, the VM features are averaged across all tasks and encoded using a shared MLP encoder to obtain the pooled VM representation $\mathbf{p}^{\text{vm, avg}}$.

These two context vectors are concatenated to form the critic input embedding as defined in eq. (3.50).

$$e^{\text{critic}} = \left[\mathbf{p}^{\text{task}} \parallel \mathbf{p}^{\text{vm, avg}} \right] \tag{3.50}$$

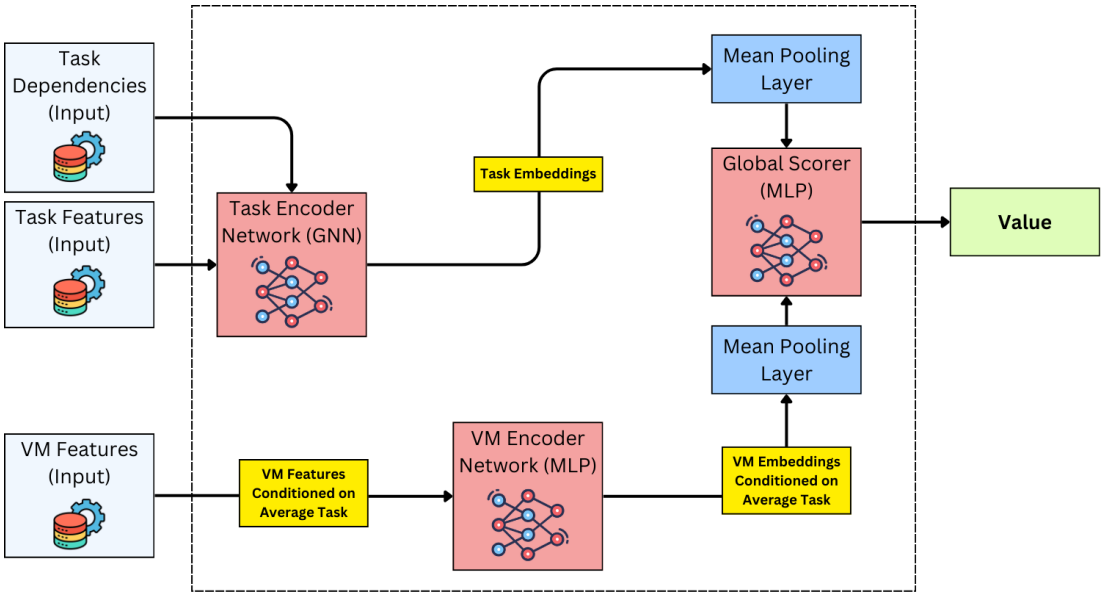

Fig. 3.13: Critic Network Architecture

This combined embedding is passed through a MLP, which outputs a scalar value prediction $V(s)$ representing the expected return from the current state.

It is important to note that the Critic network does not share parameters with the Actor networks. While the Task Actor and VM Actor share their encoder and decoder parameters internally, the encoders used by the Critic are independently parameterized. This design allows the Critic to learn a stable and dedicated representation for value estimation without being influenced by the optimization dynamics of the policy network.

3.3.7 Synthetic Dataset Generation

Following the general methodology outlined in Section 3.2.6, a new synthetic workflow dataset is generated during each training or evaluation iteration to prevent overfitting. While the underlying Erdős-Rényi $G(n, p)$ model [136] and connectivity constraints remain the same, the following parameters are adjusted for this specific agent evaluation.

Unlike the fixed node counts in the previous section, the number of tasks n in each DAG is sampled uniformly from the range $[1, N_{\text{task}}^{\text{max}}]$, with the total number of tasks across all workflows summing to N_{task}. The task length distribution follows the same normal distribution and clipping logic previously defined.

Resource allocation for this dataset incorporates additional variability. The number of VMs and physical hosts are selected from $[1, N_{\text{vm}}^{\text{max}}]$ and $[1, N_{\text{host}}^{\text{max}}]$ respectively. Furthermore, VM memory and CPU core allocations, as well as task resource requirements, are drawn uniformly maintaining the compatibility constraints between tasks and VMs.

As established in Table 3.2, host-level characteristics continue to be derived from the SPECpower benchmark suite [47], ensuring consistency in power consumption and

CPU speed ranges across all experiments.

3.3.8 Real-World Dataset Generation

Similar to synthetic dataset generation, the real-world datasets are also produced using a configurable set of parameters. However, instead of sampling task and VM characteristics from predefined distributions, these parameters are derived directly from real-world statistics, ensuring that the generated characteristics reflects realistic distributions.

For task and VM characteristic generation, the Google cluster-usage traces v3 [28] dataset was utilized. This dataset provides operational data from eight different Borg cells for the month of May 2019. Given the large size of the dataset, analysis was limited to data from the America/New York timezone, specifically tasks that started after May 1, 2019 and before May 7, 2019. To ensure manageability, 100,000 records were randomly sampled to determine task and VM statistics. Additionally, only tasks with a runtime of less than 300 s were considered, as 300 s represents the maximum measurement length. Using these extracted statistics, the configurations for task lengths, resource requirements, and VMs were derived by applying appropriate scaling factors. This adjustment was necessary because many values in the Google trace are obfuscated for privacy reasons. The final configuration parameters used for real-world dataset generation are summarized in Table 3.8.

Although the real-world dataset parameters are derived from Google cluster-usage trace statistics, the datasets themselves are generated from aggregated summaries rather than direct trace replay. Consequently, fine-grained temporal behaviors such as burstiness, peak versus off-peak workload dynamics, and complex dependency structures present in the original traces may not be fully preserved. This design choice was made to enable the generation of a large number of randomized yet statistically grounded datasets for training and evaluation, ensuring controllability and reproducibility. A sensitivity analysis (Section 4.2.3) is included to verify that the learned policies respond consistently to changes in preference weights during training, providing additional evidence of robustness across configurations. Exploring full-fidelity trace replay and richer dependency modeling remains an avenue for future work.

Host characteristics were derived from the SPECpower benchmark suite [47], as described in Section 3.2.6.

The linear-branch-merge-sink DAG structure was employed to model task dependencies. Representative examples of such DAGs are illustrated in Figure 3.14. This structure was selected for simulation as it reflects a foundational pattern commonly observed in production cloud workflows, including Extract-Transform-Load (ETL) processes, machine learning pipelines, and data orchestration systems. It effectively captures both sequential dependencies and parallel execution paths-two essential characteristics of real-world workflows. Moreover, this structure encompasses workflow

TABLE 3.8: SUMMARY OF CONFIGURATION PARAMETERS DERIVED FROM THE GOOGLE CLUSTER-USAGE TRACES FOR REAL-WORLD DATASET GENERATION

Description	**Description**
VM CPU Speed (MIPS)	Distribution of virtual machine CPU speeds derived from Google cluster traces: 1188 (17%), 1473 (∼0%), 1818 (39%), 2178 (21%), 2946 (2%), and 3072 (21%) MIPS.
VM CPU Memory (MB)	Memory capacity distribution of VMs: 683 (9%), 1024 (0%), 1366 (62%), 2048 (17%), 2732 (6%), and 4096 (6%) MB.
VM CPU Core Count	Number of CPU cores per VM with observed proportions: 1 (22%), 2 (23%), 5 (26%), 10 (20%), 15 (9%), 25 (∼0%), and 50 (∼0%).
Task Length Distribution	Task computational length modeled using a normal distribution with mean $\mu = 3{,}732.02$ and standard deviation $\sigma = 5{,}634.61$.
Priority Task Percentage	Percentage of tasks categorized as priority tasks in the generated dataset: 68.39%.

patterns 1 (Sequence), 2 (Parallel Split), and 3 (Synchronization) as defined in [137].

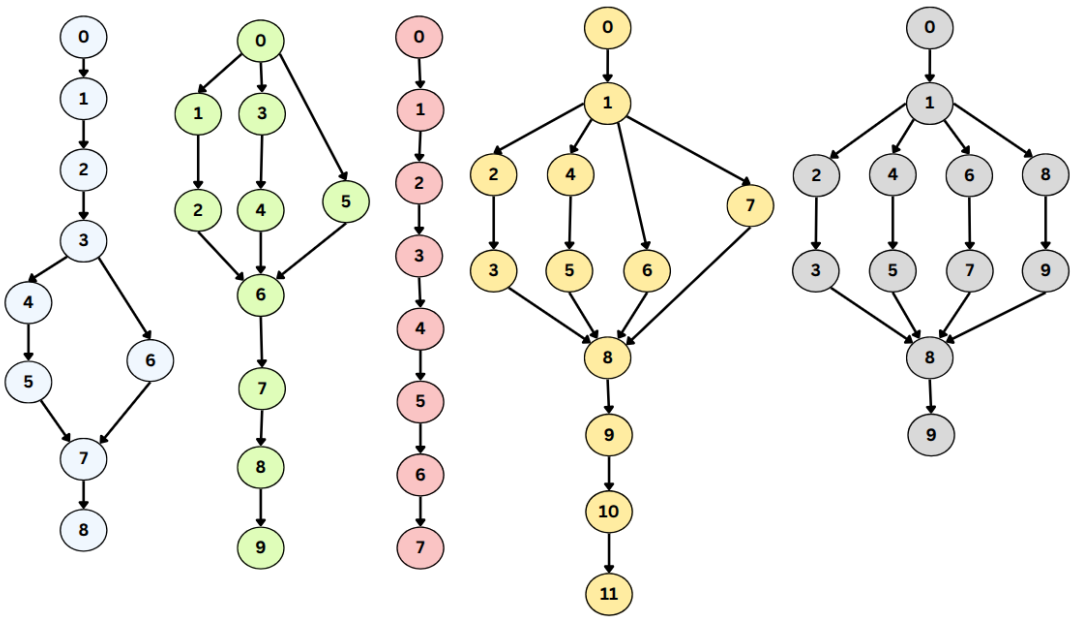

Fig. 3.14: Linear-branch-merge-sink DAG structure

Host characteristics were derived from the SPECpower benchmark suite [47], as described in Section 3.3.7.

To simulate realistic cloud environments, this study models two key aspects of

dynamic behavior: random VM breakdowns and inaccurate task length estimation. Thus, a variant of the real-world dataset configurations was used with additional dynamic properties layered on top to mimic real operational uncertainty.

VM failures may occur during operation due to various causes such as system crashes, hardware malfunctions, or scheduled maintenance [138]. To account for this, parameters were introduced to control the frequency and timing of VM failures. For simplicity, only single-VM failures are modeled, ensuring that at any given moment, no more than one VM is marked as unavailable. To avoid deadlocks or unschedulable conditions, VMs that are the sole compatible option for any task are excluded from failure events.

In practical scenarios, task execution times are often estimated prior to scheduling, but these estimates may differ from actual runtimes due to variability in data, algorithmic complexity, or system-level fluctuations [13]. To reflect this, stochastic deviations between estimated and actual task lengths are incorporated, thereby capturing the uncertainty commonly encountered in runtime predictions.

3.3.9 Tuning Reward Function Parameters

The reward function weights α, β, and γ were tuned prior to training to ensure that no reward component was disproportionately emphasized. This tuning was performed by employing a random scheduling algorithm to drive the scheduler, and collecting the reward components across multiple samples. From these observations, inverse average values were computed and used as normalization factors, allowing each reward term to contribute approximately equally under random scheduling conditions.

3.3.10 Simulation Environment

The simulation for both training and testing was conducted using a custom-built Python simulation environment implemented using NumPy and PyTorch. The environment models workflow DAG execution, resource constraints, task scheduling, and energy consumption, and supports dynamic scenarios such as VM failures and task length uncertainty. The training and all the evaluations were conducted on a laptop equipped with an Intel Core i7-14650HX CPU, 16GB of RAM, and an NVIDIA RTX 4060 Laptop GPU with 8GB VRAM, running PyTorch 2.6 with CUDA 13.0 support.

3.3.11 Agent Training

3.3.11.1 Agent Training on Synthetic Dataset

The reinforcement learning agent was trained using the PPO algorithm, implemented in Python based on the CleanRL framework [134]. Weights were initialized using the default initialization schemes provided by PyTorch/PyTorch Geometric (Kaiming

or Xavier depending on layer type). Training was performed over a fixed number of episodes, with each episode representing a complete scheduling scenario. In each episode, a new synthetic dataset was generated according to the parameters specified in Table 3.9. At the end of every episode, the agent's policy was updated using the PPO algorithm to maintain a balance between exploration and exploitation. The hyperparameters used for training are summarized in Table 3.10 and Table 3.11. For clarity and reproducibility, Table 3.10 also reports the random seed, number of parallel environments, and total training timesteps.

TABLE 3.9: TRAINING DATASET GENERATION PARAMETERS

Parameter	**Description**	**Value**
N_{task}	Number of tasks	50
$N_{\text{task}}^{\text{max}}$	Max number of tasks per workflow	10
$L_{\text{task}}^{\text{min}}$	Min task length	500
$L_{\text{task}}^{\text{max}}$	Max task length	100,000
$N_{\text{vm}}^{\text{max}}$	Max number of VMs	5
$N_{\text{host}}^{\text{max}}$	Max number of physical hosts	4
$S_{\text{vm}}^{\text{min}}$	Min CPU speed (MIPS) for VMs	500
$S_{\text{vm}}^{\text{max}}$	Max CPU speed (MIPS) for VMs	5000

TABLE 3.10: SUMMARY OF PPO HYPERPARAMETERS USED FOR TRAINING THE AGENT AND KEY REPRODUCIBILITY SETTINGS

Parameter	**Description**	**Value**
lr	Learning rate (annealed)	2.5×10^{-4}
N	Batch size	512
γ	Discount factor	0.99
λ	Generalized Advantage Estimation (GAE) factor	0.95
ϵ	Clipping parameter	0.2
K	Number of epochs per PPO update	4
β	Entropy coefficient	0.01
c	Maximum gradient norm for clipping	0.5
B	Number of steps per policy rollout	128
M	Number of minibatches per update	4
θ_{te}	Task encoder (GAT) hidden layers	64 (3 layers)
θ_{ve}	VM encoder hidden layers	64 (2 layers)
θ_{d}	Task/VM decoder hidden layers	64 (2 layers)
θ_{gs}	Critic global scorer hidden layers	64 (2 layers)
E	Number of features in embeddings	8
	Random seed	0
	Total timesteps	200,000
	Early stopping criteria	None
	Number of parallel environments	4

TABLE 3.11: REWARD SCALING FACTORS USED FOR TRAINING THE AGENT ON THE SYNTHETIC DATASET

Parameter	Description	Value
α	Weight of makespan reward	0.541
β	Weight of energy consumption reward	9.093
γ	Weight of QoS penalty score reward	0.425

Figure 3.15 illustrates the training curve of the agent with equal preference weights ($\lambda_1 = \lambda_2 = \lambda_3 = 1$), showing the increase in episodic return as training progressed. The training results indicate that the PPO algorithm, in conjunction with the GAT-based architecture, effectively learned to minimize the makespan and energy consumption while handling dynamic task arrivals and dependencies.

Fig. 3.15: Episodic return during training of the agent with equal weights

Additionally, agents were trained under various preference configurations. Representative training curves for some of these agents are illustrated in Figure 3.16.

3.3.11.2 Agent Training on Real-World Dataset

To demonstrate the benefits of fine-tuning and training the agent on real-world scenarios, additional training was performed using the real-world dataset described in Section 3.3.8. The number of tasks (N_{task}), maximum number of tasks per workflow ($N_{\text{task}}^{\text{max}}$), maximum number of VMs ($N_{\text{vm}}^{\text{max}}$), and maximum number of hosts ($N_{\text{host}}^{\text{max}}$) were kept as specified

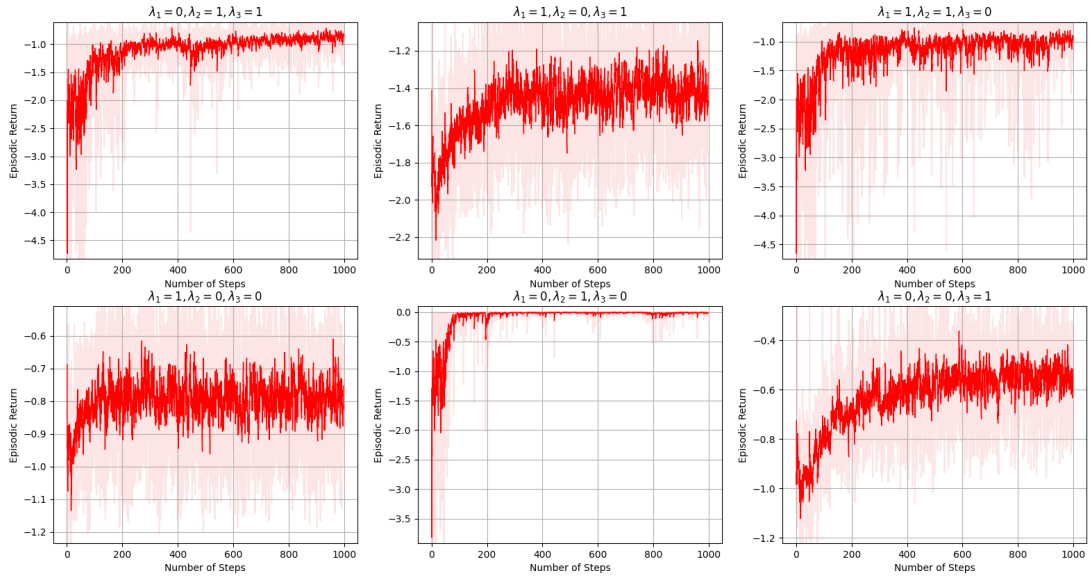

Fig. 3.16: Episodic return during training of the some other agents

in Table 3.9. All training parameters as well were retained as specified in Table 3.10, except for the reward scaling factors α, β, and γ. These parameters were re-tuned using the method outlined in Section 3.3.9, and their updated values are provided in Table 3.12.

TABLE 3.12: REWARD SCALING FACTORS USED FOR TRAINING THE AGENT ON THE REAL-WORLD DATASET

Parameter	**Description**	**Value**
α	Weight of makespan reward	0.91
β	Weight of energy consumption reward	14.398
γ	Weight of QoS penalty score reward	0.602

3.3.12 Agent Evaluation

For evaluating the RL agent, a synthetic dataset was generated with attributes given in table 3.13. During evaluation, the maximum number of workflows and tasks is varied to assess the model's performance across several scenarios.

In addition, evaluation was performed using a real-world dataset, denoted as DS^{real}. This dataset was derived from the method described in Section 3.3.8. The parameters used to generate this dataset are summarized in Table 3.14.

An extended variant of this dataset, denoted as DS^{dyn}, was also used to simulate dynamic execution conditions, incorporating random VM breakdowns and inaccurate task length estimations. The specific parameters used to introduce these dynamic behaviors are listed in Table 3.15.

TABLE 3.13: EVALUATION SYNTHETIC DATASET GENERATION PARAMETERS

Parameter	Description	$\mathbf{DS}_1^{\mathbf{syn}}$	$\mathbf{DS}_2^{\mathbf{syn}}$	$\mathbf{DS}_3^{\mathbf{syn}}$
N_{task}	Number of tasks	50	250	1000
$N_{\text{task}}^{\text{max}}$	Max number of tasks per workflow	10	20	50
$L_{\text{task}}^{\text{min}}$	Min task length	500	250	1
$L_{\text{task}}^{\text{max}}$	Max task length	100,000	200,000	500,000
$N_{\text{vm}}^{\text{max}}$	Max number of VMs	5	20	50
$N_{\text{host}}^{\text{max}}$	Max number of physical hosts	4	5	10
$S_{\text{vm}}^{\text{min}}$	Min CPU speed (MIPS) for VMs	500	250	100
$S_{\text{vm}}^{\text{max}}$	Max CPU speed (MIPS) for VMs	5000	10,000	20,000

TABLE 3.14: EVALUATION REAL-WORLD DATASET GENERATION PARAMETERS

Parameter	Description	Value
N_{task}	Number of tasks	250
$N_{\text{vm}}^{\text{max}}$	Max number of VMs	20
$N_{\text{host}}^{\text{max}}$	Max number of physical hosts	5

TABLE 3.15: EVALUATION DYNAMIC REAL-WORLD DATASET GENERATION PARAMETERS

Parameter	Description	Value
N_{task}	Number of tasks	250
$N_{\text{vm}}^{\text{max}}$	Max number of VMs	20
$N_{\text{host}}^{\text{max}}$	Max number of physical hosts	5
$T_{\text{downtime}}^{\text{max}}$	Max VM downtime duration	20
$T_{\text{tbf}}^{\text{max}}$	Max time between failures	40
$E_{\text{len}}^{\text{max}}$	Max error rate of the task length estimation	0.1

3.3.13 Comparison Algorithms

2SD-GAT algorithm is evaluated against a diverse set of baseline algorithms to comprehensively assess its performance across multiple objectives and scheduling scenarios. Table 3.16 summarizes the comparison algorithms, which span both static and dynamic scheduling strategies, as well as single-objective and multi-objective optimization approaches.

Among the single-objective static algorithms, HEFT and FERPTS represent well-established heuristic methods that have been widely adopted in workflow scheduling due to their efficiency in minimizing makespan. Min-Min and Max-Min heuristics offer contrasting strategies by prioritizing tasks with the shortest and longest execution times, respectively. These methods serve as useful baselines for evaluating fundamental scheduling behavior.

For dynamic single-objective scheduling, algorithms such as Round-Robin, Least-Loaded-First, Random, and Weighted Dynamic Scheduling are included. These operate in an online manner and reflect practical strategies used in real-time cloud environments, particularly under uncertainty and varying workloads.

To benchmark against state-of-the-art multi-objective optimization techniques, the evaluation includes MOHEFT, NSGA-II, NSGA-III, and MOEA/D. These evolutionary algorithms are designed to approximate the Pareto front across conflicting objectives such as makespan, energy consumption, and QoS penalties. They provide a strong comparative foundation to assess the effectiveness of 2SD-GAT in producing diverse and high-quality trade-offs.

In addition, GNN-Flat, the algorithm from Section 3.2, is included as a learning-based baseline to provide a closer architectural comparison to 2SD-GAT. GNN-Flat was retrained using the same dataset and training budget to ensure a fair comparison. However, due to its simpler architecture and flat action space, it did not fully converge when optimizing all three objectives simultaneously within the given training budget (200,000 timesteps). Including GNN-Flat nonetheless offers insight into how architectural and action-space design choices influence multi-objective learning performance.

All algorithms were tested on the same datasets (DS_1^{syn}, DS_2^{syn}, DS_3^{syn}, DS^{real}, DS^{dyn}) under identical experimental conditions. The DRL methods (GNN-Flat and 2SD-GAT) were trained using the same datasets and simulation environment (Sections 3.3.11.1 and 3.3.11.2). To maintain fairness, the same input features and system-level information were provided to the baseline algorithms wherever applicable.

3.3.14 Evaluation Metrics

The results of 2SD-GAT algorithm are compared with the baseline methods described in Section 3.3.13. Due to the multi-objective nature of the problem, simple scalar comparisons of makespan, energy consumption, or QoS penalties are insufficient. Instead, standard metrics from MOO are employed to evaluate the performance of each algorithm.

One of the primary metrics used is Hypervolume, which measures the volume of the objective space dominated by the obtained Pareto front and bounded by a predefined reference point [31]. A higher hypervolume value indicates a better trade-off coverage among objectives. The reference point is chosen such that it is dominated by all solutions across all evaluated methods, ensuring fair comparison. The formal definition of HV is provided in eq. (3.51).

$$HV(S, r) = \text{volume}\left(\bigcup_{x \in S} [f_1(x), r_1] \times [f_2(x), r_2] \times \cdots \times [f_m(x), r_m]\right) \tag{3.51}$$

TABLE 3.16: SUMMARY OF COMPARISON ALGORITHMS

Algorithm	**Description**
HEFT [6]	Heterogeneous Earliest Finish Time algorithm; prioritizes tasks and assigns them to VMs to minimize earliest finish times.
FERPTS [6]	Fast and Energy-Aware Resource Provisioning and Task Scheduling; a static energy-aware workflow scheduling method.
Random	Randomly assigns tasks to VMs at runtime.
Round Robin [135]	Assigns tasks to VMs in cyclic order regardless of task or VM characteristics.
Least-Loaded-First	Assigns tasks to the VM with the least current workload.
Weighted Dynamic	Dynamically assigns tasks to minimize a weighted score of all the objectives.
MOHEFT [20]	Multi-objective extension of HEFT producing Pareto-optimal schedules for multiple objectives.
NSGA-II [97]	Non-dominated Sorting Genetic Algorithm II; evolutionary algorithm for approximating Pareto fronts (Population size = 100, Generations = 500).
NSGA-III [98]	Improved variant of NSGA-II suitable for many-objective problems, using reference directions for diversity preservation (Population size = 100, Generations = 500, Ref Direction = Uniform 12 Partitions).
MOEA/D [99]	Multi-objective Evolutionary Algorithm based on Decomposition; decomposes a MOO problem into scalar sub-problems (Generations = 500, Ref Direction = Uniform 12 Partitions).
GNN-Flat	The algorithm from Section 3.2 originally optimized for makespan and energy consumption, and retrained for makespan, energy consumption, and QoS objectives, under the same dataset and training budget as 2SD-GAT.

Here, S is the obtained Pareto front, r is the reference point, and $f_i(x)$ denotes the i-th objective value of solution x.

Another widely adopted metric is the Inverted Generational Distance, which evaluates both convergence and diversity of the Pareto front [31]. IGD calculates the average distance from each point in a known reference Pareto front to the closest point in the evaluated solution set. A lower IGD indicates that the solution set is closer to the true Pareto front. The IGD is defined in eq. (3.52).

$$IGD(P^*, S) = \frac{1}{|P^*|} \sum_{x^* \in P^*} \min_{x \in S} \|x - x^*\| \tag{3.52}$$

Here, P^* is the reference Pareto front formed by aggregating all non-dominated

solutions from all algorithms, and S is the solution set being evaluated.

In addition to these Pareto-based indicators, Decision Latency is also considered, which is the time taken to make a single scheduling decision. Since this study focuses on dynamic scheduling scenarios, it is critical that scheduling decisions are computed in real-time or near-real-time. Therefore, lower decision latency reflects better practical applicability of the method in real-world cloud systems.

Lastly, to analyze the dominance relationships between algorithms, the Pareto fronts produced by each method are compared. Given the three-objective setting, 2D projections of the Pareto front are presented for all three objective pairs to illustrate trade-offs and dominance across different dimensions.

3.4 Apache Airflow Integration

The first workflow engine selected for integration was Apache Airflow due to its widespread adoption in both scientific workflows and commercial data processing pipelines.

3.4.1 Airflow Architecture Overview

In a standard Airflow deployment, workflows are defined as DAG files that specify task dependencies and execution rules. The architecture consists primarily of the following components:

- **Scheduler**: Responsible for parsing DAG files, determining task dependencies, and scheduling tasks for execution based on defined triggers and priorities.
- **Executor**: Executes scheduled tasks and can be configured to run locally, on a cluster, or using backends such as Celery or Kubernetes.
- **Worker**: The process that runs task code when invoked by the executor.
- **Metadata Database**: Stores the state of DAGs, tasks, and scheduling decisions, enabling tracking and management of workflow execution.
- **Web Server**: Provides a user interface for monitoring and managing workflows, including DAG visualization and task status inspection.

As illustrated in the Airflow architecture (Figure 3.17), the Airflow Scheduler monitors task completion, evaluates dependency constraints defined in DAG files, and dispatches runnable tasks either directly to workers or to a shared queue, depending on the executor configuration. When a queue-based executor is used, workers independently retrieve tasks from the queue for execution; otherwise, tasks are executed directly by the executor.

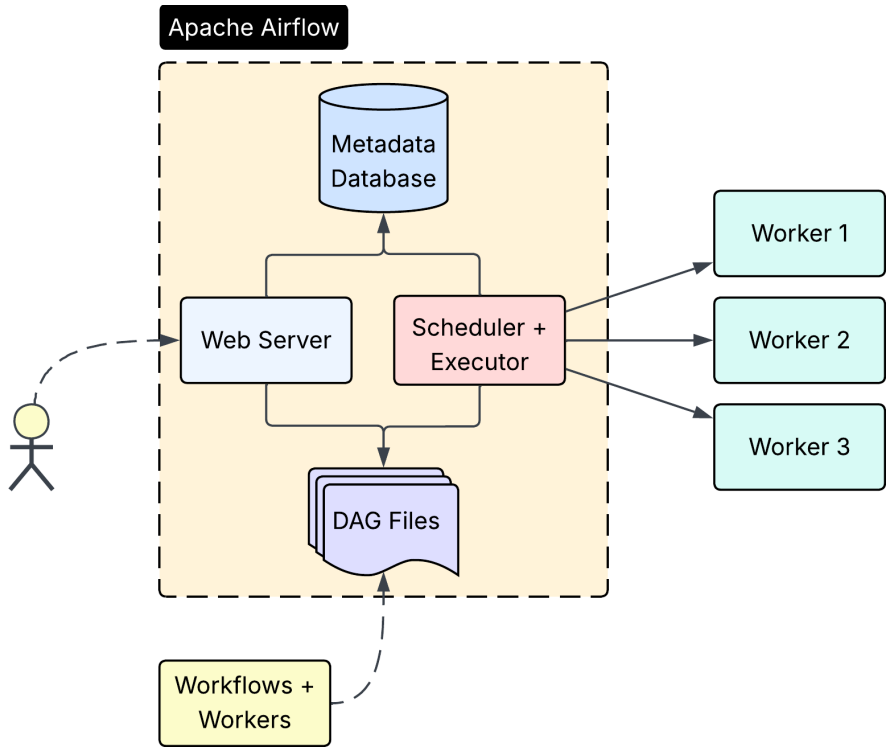

Fig. 3.17: Apache Airflow Architecture (Local Executor)

This design ensures scalability and reliability; however, execution decisions are derived primarily from predefined DAG logic rather than adaptive runtime optimization.

3.4.2 Challenges in Integrating Adaptive Scheduling

During integration, several architectural constraints of Airflow were identified that directly influenced the design of the proposed system.

- **Static DAG Definition**: Airflow requires workflows to be predefined as static DAG files prior to execution. Consequently, scheduling decisions must largely be determined before runtime begins. This batch-oriented execution model limits the introduction of dynamic or adaptive scheduling behaviour, as task placement decisions cannot easily evolve during execution.
- **Batch-Oriented Execution Model**: Scheduling decisions are evaluated primarily at DAG parsing time rather than continuously optimized during execution. While this improves stability and reproducibility, it restricts online learning-based scheduling approaches that rely on runtime feedback.
- **Tight Integration with Executors**: Airflow's execution mechanism is tightly coupled with its executor framework. Introducing custom "decide-where-to-run" logic would normally require deep modifications to Airflow's core scheduler and

executor components. Such modifications reduce maintainability and compatibility with future Airflow versions, making direct alteration undesirable.

- **Deterministic Reproducibility**: Airflow emphasizes deterministic execution for logging, retries, and historical reproducibility. Any adaptive scheduling mechanism must therefore preserve deterministic task ordering to maintain Airflow's operational guarantees.

These constraints motivated the design of a plugin-based integration strategy, enabling intelligent scheduling without modifying Airflow's internal scheduler.

3.4.3 Plugin-Based Scheduling Architecture

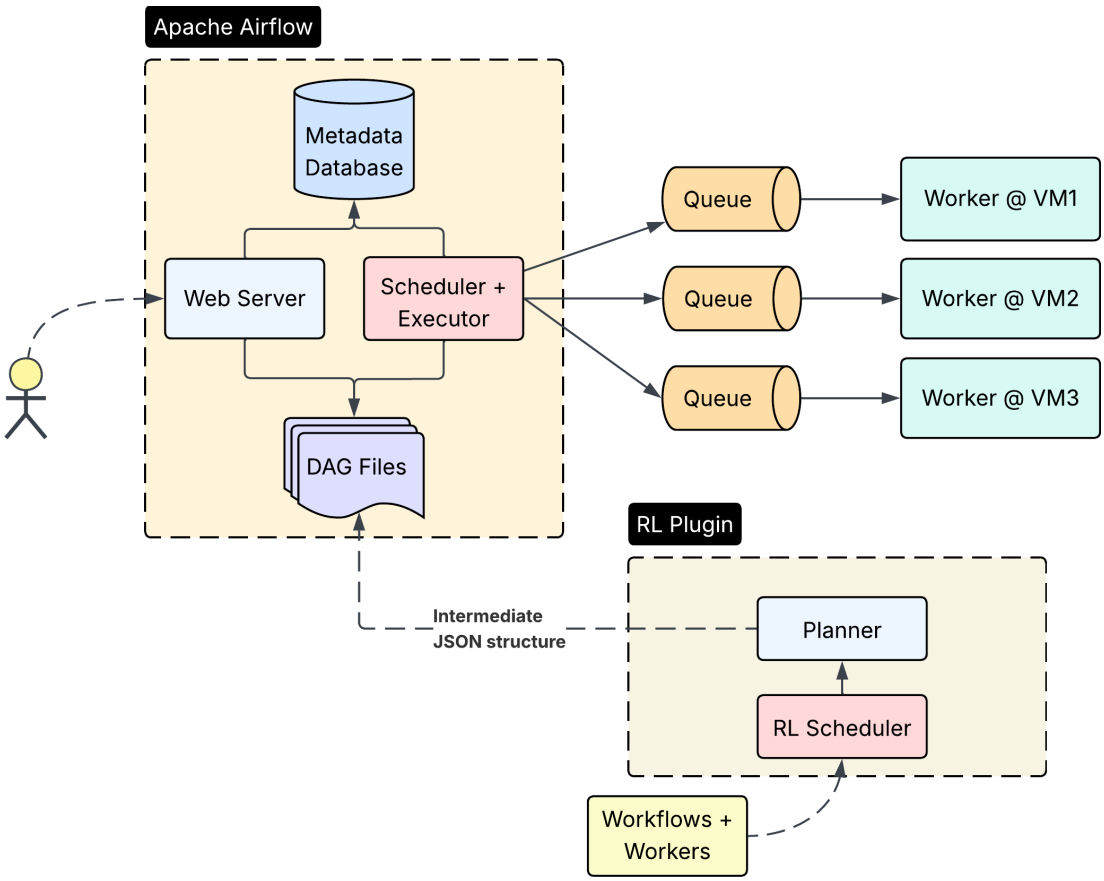

Fig. 3.18: Modified Apache Airflow Architecture (Celery Executor with RL-Driven Scheduling)

To overcome the identified limitations, an external planning layer was introduced during workflow definition. As shown in Figure 3.18, the enhanced architecture augments the standard Airflow workflow with a RL-driven scheduling pipeline that operates prior to workflow execution while remaining compatible with Airflow's static DAG model.

In this approach, workflow and worker information are provided explicitly as user inputs, replacing the scheduling logic that was previously embedded directly within DAG files. Using this information, the RL scheduler performs a simulation of workflow execution based on the known characteristics of the system and the workflow. The scheduler then generates an optimized execution plan that introduces additional dependency edges within the DAG to enforce the execution order determined by the learned policy. Furthermore, the plan includes queue assignments for each task, enabling controlled worker selection during execution.

The resulting scheduling decisions are transformed into an intermediate JSON representation that captures task dependencies, worker mappings, and execution constraints in a format suitable for Airflow ingestion. A specialized DAG file, implemented as a plugin to Airflow, reads this generated JSON plan and constructs the executable workflow accordingly.

To enforce task placement without modifying Airflow's internal components, the execution environment is configured such that each VM is associated with a dedicated queue. The generated DAG assigns tasks to these queues according to the RL scheduler's decisions, allowing Airflow's native scheduling mechanism to dispatch tasks to workers indirectly based on queue selection. Consequently, Airflow executes the workflow according to the optimized plan while preserving its original scheduling infrastructure and deterministic execution guarantees.

Instead of altering Airflow's scheduler, the proposed system influences execution indirectly by controlling task dependencies and queue assignments within the DAG itself.

3.4.4 Execution and Evaluation Setup

For evaluation purposes, workflow datasets derived from DS^{real} were generated to simulate dependency-aware cloud workloads. To maintain a manageable experimental environment, the number of tasks, VMs, and hosts was reduced to 50, 10, and 3 respectively, while preserving the original task length distributions and dependency structures. This reduction enabled controlled evaluation of scheduling decisions while still retaining characteristics representative of real-world workflows.

Multiple containerized environments were configured to emulate independent virtual machines. The evaluation considered three performance metrics across all experiment runs: Execution Time, QoS Penalty, and Energy Consumption. In total, 10 experiment runs were conducted using different datasets generated according to the procedure described in Section 3.3.8.

Apache Airflow was executed under two configurations: (1) Baseline Mode, using Airflow's default scheduler with the original DAG files, and (2) RL-Driven Mode, using DAG files generated by the proposed RL-driven planning framework. Figure 3.19 shows

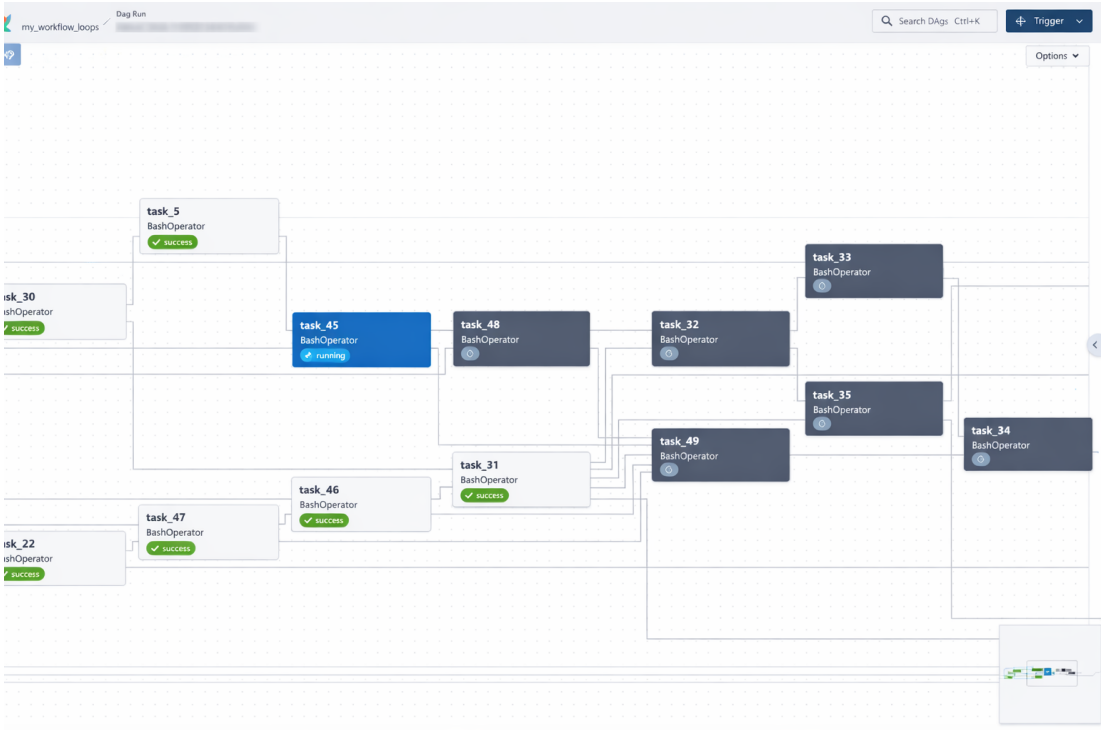

Fig. 3.19: Apache Airflow executing workflows in RL-Driven mode

a screenshot of Apache Airflow executing the workflows in RL-Driven mode.

3.5 Integration with Kubernetes Scheduling

Following the integration with Apache Airflow, the next phase focused on incorporating the proposed RL-based scheduler into a Kubernetes-based execution environment. While Kubernetes is widely used for container orchestration in cloud-native systems, it does not natively provide workflow semantics. Workflow execution in Kubernetes is typically supported through higher-level frameworks such as Argo Workflows [18], which define workflows declaratively using YAML manifests. Since Argo workflows ultimately translate execution logic into Kubernetes resource definitions, it could theoretically be integrated using an approach similar to the one discussed on Section 3.4. However, in this section, the focus was shifted toward integrating directly with Kubernetes itself, without relying on Argo, in order to evaluate scheduling decisions at the infrastructure level.

3.5.1 Existing Kubernetes Scheduling Architecture

Kubernetes follows a declarative resource management model in which applications are deployed as containers grouped into units called pods. Users submit pod specifications to the Kubernetes API server using YAML manifests that describe resource requirements,

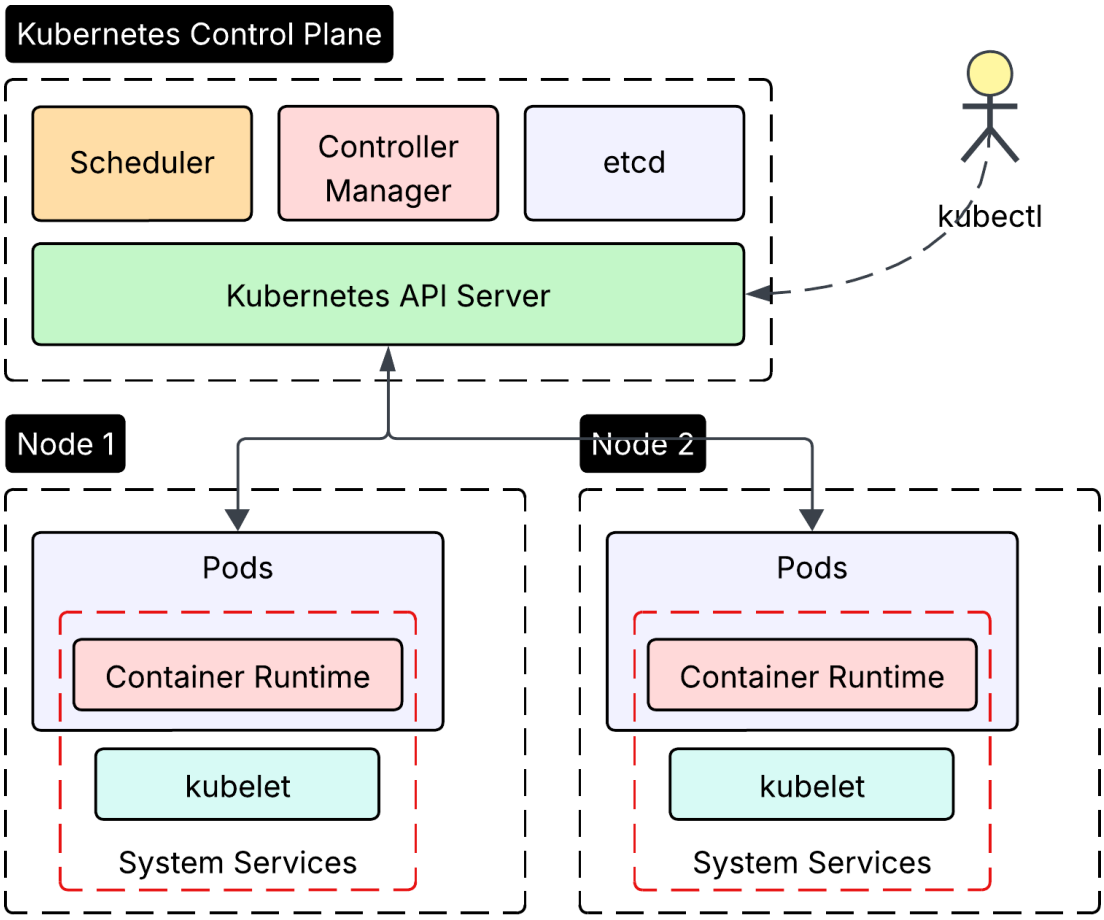

Fig. 3.20: Kubernetes architecture overview with native scheduling components

execution parameters, and deployment constraints. Once submitted, pods enter the cluster lifecycle and are managed by the Kubernetes control plane.

In the default architecture, newly created pods are initially placed in a `PENDING` state until they are assigned to a suitable node. The native Kubernetes Scheduler continuously monitors unscheduled pods and evaluates available cluster nodes based on resource availability, placement constraints, affinity rules, and scheduling policies. Modern Kubernetes scheduling follows a multi-stage Scheduling Framework consisting of filtering, scoring, and binding phases. During filtering, nodes that cannot satisfy resource or policy requirements are eliminated. During scoring, remaining nodes are ranked according to predefined heuristics such as load balancing and resource utilization. In the binding phase, the scheduler finalizes placement decisions through pre-bind and post-bind operations, where additional actions such as volume attachment and network configuration may be completed before the pod is bound to the selected node via the Kubernetes API server.

Although this architecture is highly scalable and effective for independent microservice workloads, scheduling decisions are typically evaluated on a per-pod basis by the default scheduler. Advanced scenarios such as high-performance computing or AI

workloads requiring coordinated execution are supported through extensions of the Scheduling Framework, including scheduling gates, coscheduling plugins, and pod groups that allow multiple pods to be held in a waiting state until group-level constraints are satisfied.

Kubernetes does not natively define a workflow abstraction; however, workflow semantics can be introduced within the Kubernetes API through Custom Resource Definitions (CRDs) and controllers such as Argo Workflows or Tekton, which translate DAG structures into controlled pod creation sequences. At the core level, dependency-aware execution must therefore be enforced by higher-level controllers or external scheduling logic rather than the native scheduler itself. The only native mechanism for tightly grouping containers during placement is the Pod abstraction, where multiple containers (including sidecars) share the same scheduling lifecycle and are always co-located on the same node.

Furthermore, the default scheduler primarily relies on heuristic-based placement strategies designed for general cluster utilization rather than application-specific optimization. This limits its ability to incorporate advanced decision-making mechanisms such as learning-based scheduling or multi-objective optimization. These limitations motivate the introduction of an external scheduling component capable of reasoning about workflow dependencies and resource selection simultaneously, as described in the Section 3.5.2.

3.5.2 Proposed Kubernetes-Based Scheduling Architecture

To enable intelligent scheduling within Kubernetes, a specialized scheduler pod was deployed as part of the cluster. This pod encapsulates the custom RL scheduler and operates alongside the native Kubernetes control plane. When a workflow execution begins, all workflow-related pods are initially deployed in a `PENDING` state. This design prevents immediate scheduling by the default Kubernetes scheduler and allows the custom scheduling component to take control of placement decisions.

The RL scheduler continuously queries the Kubernetes API server to monitor all pending pods whose dependency constraints have been satisfied. Information about pod requirements, node characteristics, and workflow state is collected and forwarded to the trained model. Based on the model's output, the scheduler selects an appropriate node for execution and explicitly binds each pod to the chosen node through the Kubernetes API. This binding operation effectively determines the VM or worker on which the pod will execute.

Unlike Airflow, Kubernetes does not inherently support DAG-style execution constraints. To preserve workflow dependencies, the scheduler maintains an internal representation of workflow state and ensures that pods are only considered for placement once their prerequisite tasks have completed.

3.5.3 Execution Flow

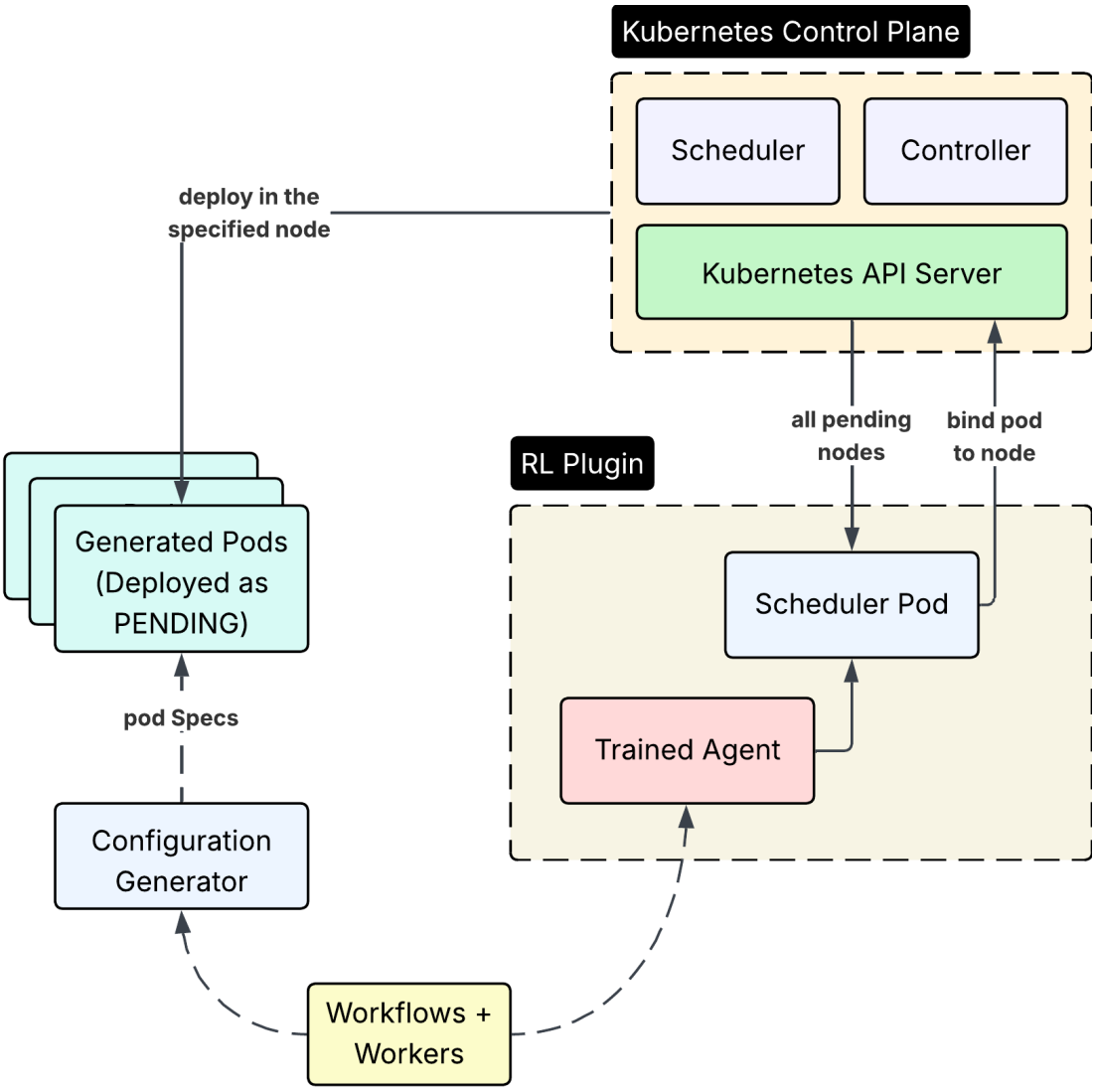

Fig. 3.21: Kubernetes-based scheduling architecture with RL-driven scheduler pod

As illustrated in Figure 3.21, workflow specifications are first transformed into Kubernetes pod configurations by a configuration generator that produces YAML manifests describing task execution requirements. The configuration generator is responsible for generating YAML configurations for each task, capturing execution parameters and resource specifications required for deployment within the cluster. These pods are submitted to the Kubernetes API server and remain in a `PENDING` state until explicitly scheduled.

A dedicated Python Scheduler Pod operates as the core orchestration component of the proposed system. This scheduler continuously queries the Kubernetes API to identify all pending pods whose dependency pods have already completed, thereby preserving workflow execution order. The scheduler maintains an internal scheduler

state that tracks the status of the currently executing workflow as well as the state of available nodes and previously assigned pods, allowing consistent and dependency-aware decision making.

The scheduler interacts with a trained RL agent based on 2SD-GAT that encodes task and node features using a graph neural network representation. The agent performs two-stage decision-making, jointly determining task selection and virtual machine (node) assignment based on learned policies. Guided by the agent's recommendations, the scheduler selects an appropriate pod and binds it to a specific node through the Kubernetes API server, effectively determining where the task will execute. Once binding is completed, Kubernetes proceeds with normal container deployment and execution using its native lifecycle management mechanisms.

This approach enables the RL scheduler to control placement decisions while remaining fully compatible with Kubernetes' native resource orchestration framework, ensuring that DAG dependencies are respected while intelligent placement decisions are enforced externally.

3.5.4 Execution and Evaluation Setup

Evaluation of the Kubernetes-based scheduling framework was conducted using Kubernetes in Docker (KinD) [139], which enables the simulation of a local Kubernetes cluster composed of Docker-based container nodes. Since the native Kubernetes scheduler does not support dependency-aware workflow scheduling, comparisons were performed against several baseline heuristic schedulers rather than the default scheduler.

A methodology similar to the Airflow evaluation was followed, where workflow specifications were generated from DS^{real} and reduced to 50 tasks, 10 VMs, and 3 hosts. This configuration maintained a manageable experimental environment while preserving key characteristics of real-world workloads.

As all experiments were executed within a local cluster using relatively simple workflows, the energy consumption metric was excluded from evaluation. Consequently, performance analysis focused primarily on execution time.

3.6 Implementation of the RL-Based Pluggable Scheduler Prototype

This section describes the implementation of a custom tool developed to evaluate RL-based workflow scheduling in a controlled experimental environment. The prototype implements an event-driven scheduling framework in which a pluggable scheduling strategy selects task-worker assignments during workflow execution. The primary objective of the system is to provide a modular platform for integrating and evaluating an RL inference policy while isolating scheduling behavior from execution-specific concerns. The implementation is designed as a research prototype rather than a production system

and therefore prioritizes architectural clarity, observability, and experimental flexibility.

3.6.1 System Overview

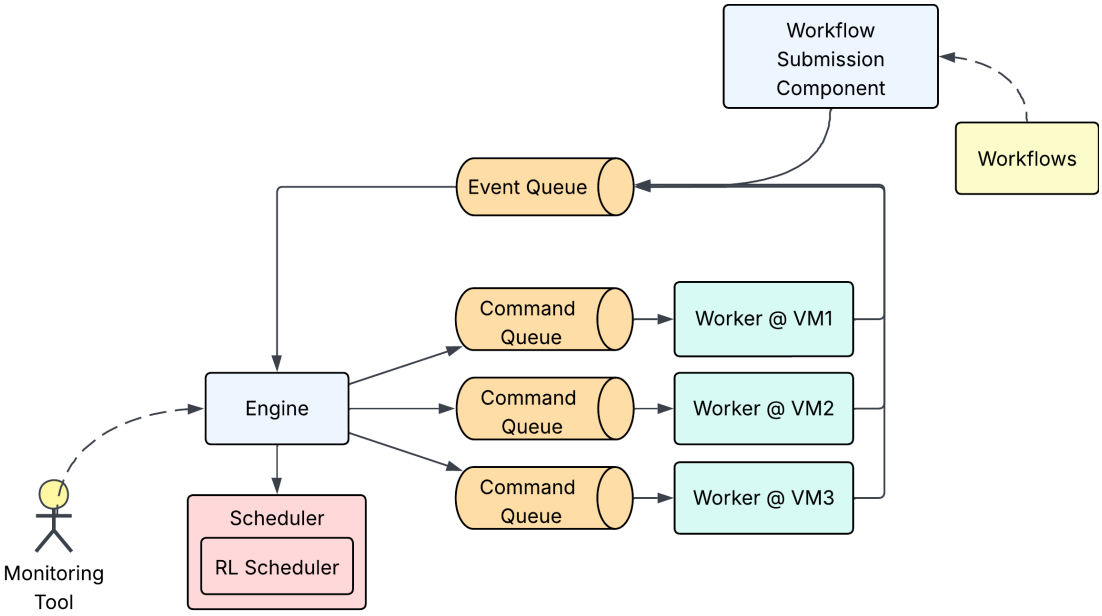

Fig. 3.22: Architecture of the RL-Based Pluggable Scheduler Prototype (Using queue-based connectors for communication)

The system targets event-driven scheduling of dependency-aware workflows executed across distributed workers. Tasks belong to workflows represented as DAGs, and scheduling decisions determine which worker executes each task once its dependencies have completed. The scheduling strategy is implemented as a pluggable module, allowing different policies to be evaluated under identical execution conditions. In this work, the strategy is assumed to be an RL-based inference policy that selects the next $(worker, task)$ assignment based on the current system state.

The architecture consists of three asynchronous services that communicate through a message broker. A central *engine* maintains workflow state and performs scheduling decisions, distributed *workers* execute tasks and report status updates, and a *submission component* injects workflow definitions into the system. Message transport is abstracted through a connector interface, allowing the underlying communication mechanism to be replaced without modifying scheduling or execution logic. This separation enables experimentation with scheduling policies independently from infrastructure concerns.

3.6.2 Core Components and Data Models

The engine acts as the central coordinator responsible for maintaining the global view of the system. It tracks the state of all tasks and workers in memory and processes incoming

events such as worker registrations, worker updates, task completions, and workflow submissions. Dependency tracking is performed explicitly so that tasks become eligible for execution only after all prerequisite tasks have completed. Whenever schedulable tasks exist, the engine invokes the scheduling strategy to determine the next assignment and issues execution commands to the selected worker.

Workers simulate task execution and periodically communicate their status to the engine. Upon completion, workers emit execution results, including estimated energy consumption and execution metadata, allowing the scheduler to incorporate performance feedback.

Workflow submission is handled by a dedicated component that parses workflow specifications expressed in JSON format and converts them into internal task definitions with explicit dependency relationships. These definitions are then submitted to the engine as workflow events, triggering the scheduling process.

Task definitions describe identifiers, priorities, dependency relationships, estimated execution length, and resource requirements. Runtime task state captures schedulability, execution status, timing metadata, energy consumption, and quality-of-service penalties. Worker descriptions encode computational characteristics such as processing speed, available resources, and estimated power consumption. All interactions between components are expressed as structured events and commands that can be serialized and transmitted through the messaging layer.

3.6.3 Execution Flow

Execution begins when a workflow specification is submitted to the system. Tasks are expanded into internal representations and registered with the engine along with their dependency constraints. Workers subsequently register themselves and provide periodic updates describing their capabilities and current load.

The engine operates an event-driven scheduling loop. Incoming events update internal state, dependency resolution marks tasks as schedulable when prerequisites complete, and the scheduling strategy is invoked whenever new scheduling opportunities arise. The selected assignment results in a start command sent to a worker, which simulates execution and returns a completion event upon finishing the task. Each completion triggers further dependency resolution and may enable additional tasks to become schedulable. Scheduling is therefore continuously re-evaluated in response to both events and periodic triggers, allowing dynamic adaptation to changing system conditions.

3.6.4 RL-Based Scheduling Integration

The scheduling strategy is implemented as an RL inference module that maps the current system state to an action. Conceptually, the state representation includes

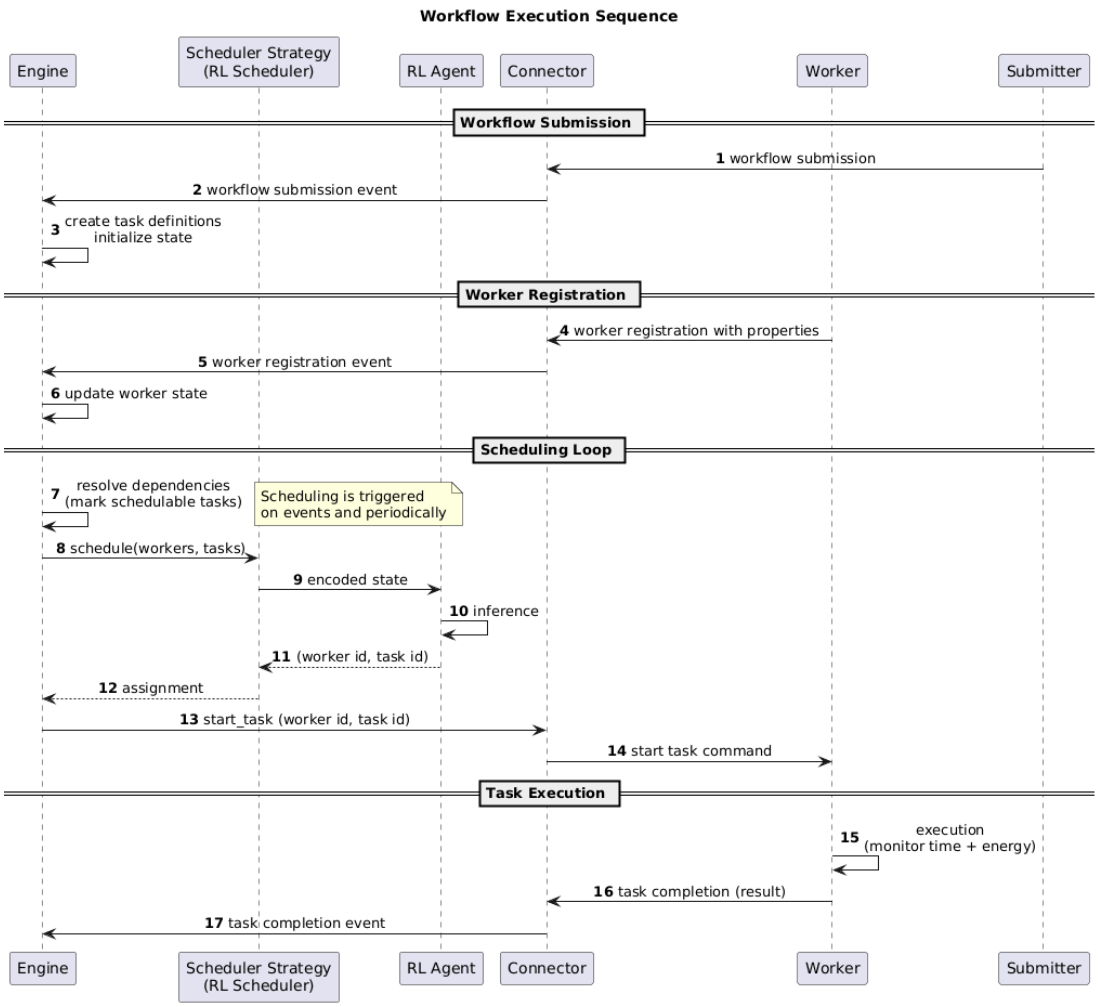

Fig. 3.23: Sequence diagram illustrating the execution flow of the RL-Based Pluggable Scheduler Prototype

features describing queued tasks, worker characteristics, and system-level performance indicators. Task features may include priority, remaining dependencies, and estimated execution length, while worker features capture computational capacity, available resources, and current utilization. Global metrics such as elapsed execution time, accumulated energy consumption, and quality-of-service penalties provide additional context for decision making.

The action space consists of selecting a valid $(worker\_id, task\_id)$ pair among all schedulable tasks and available workers. The RL policy evaluates candidate assignments and returns the action with the highest predicted value. Although training is external to this prototype, the integration point assumes an inference interface that can be invoked directly by the engine. This modular design allows the RL scheduler to be replaced by heuristic or rule-based strategies without modifying other system components, enabling fair comparative evaluation.

3.6.5 Design Rationale

Several design decisions were made to support experimental flexibility. An event-driven architecture decouples the engine from workers, enabling asynchronous execution and simplifying scalability experiments. The pluggable scheduler interface isolates decision logic, allowing rapid experimentation with alternative scheduling policies. Explicit dependency tracking guarantees correct execution order for DAG-based workflows, while the asynchronous runtime enables concurrent handling of events and scheduling triggers.

A lightweight execution model was intentionally adopted so that experimental outcomes primarily reflect scheduling behavior rather than workload variability. Continuous visualization further improves observability, allowing researchers to inspect execution patterns and verify qualitative differences between scheduling strategies.

3.6.6 Prototype Limitations and Extensibility

As a research prototype, the implementation makes several simplifying assumptions. System state is maintained entirely in memory without persistence, and fault tolerance mechanisms such as retries or message durability are not included. The architecture assumes a single scheduling engine and does not implement leader election or distributed coordination. Security or multi-tenant isolation concerns are outside the scope of this work.

Despite these limitations, the design remains highly extensible. Alternative communication backends can be introduced by implementing the connector abstraction, new scheduling strategies can be added without modifying engine logic, and the simulated worker model can be replaced with real execution environments while preserving the same command and event interfaces. This flexibility allows the prototype to serve as a foundation for future experimentation with learning-based scheduling techniques in distributed workflow systems.

3.7 Tool Development

This section describes the development of the auxiliary tools that support the pluggable scheduler prototype: workflow submission, worker execution with energy measurement, engine implementation, and DRL scheduler implementation. The source code for this tool is available in the public repository at https://github.com/kdsuneraavinash/prototype-workflow-engine.

The development used Python 3.11 with asyncio for concurrency, Pydantic for data validation, and pyRAPL for energy measurement. The tools are designed to be modular and extensible, allowing for future integration of additional scheduling strategies or measurement backends. The agent development for the DRL scheduler is also included,

showcasing the GNN-based policy architecture (2SD-GAT) used with the tool. This used PyTorch for the neural network implementation and training. PyTorch Geometric was used for the GAT layers in the agent architecture. The training process, which is outside the scope of this section, involved reinforcement learning with a custom environment implemented with OpenAI Gym interfaces.

3.7.1 Workflow Submission

Workflow submission uses a JSON schema validated by Pydantic models. The schema captures task information used to build runnable task definitions.

Listing 3.1: Tool input schema models (excerpt).

```
class TaskInput(BaseModel):
    id: int
    """unique within workflow"""

    command: str | None
    """optional command to run, defaults to a simple NOP if
       not provided"""

    priority: int
    """priority of the task, higher means more important"""

    dependencies: list[int]
    """list of task IDs that depend on this task, used to
       build the dependency graph"""

    estimated_length_mi: int
    """estimated length in million instructions (MI) for
       scheduling purposes"""

    req_memory_gb: int
    """estimated memory requirement in GB for scheduling
       purposes"""

    req_core_count: int
    """estimated core count requirement for scheduling
       purposes"""
```

The submission tool reads a workflow JSON file, validates it, expands tasks into engine-ready definitions, and emits a `WorkflowSubmissionEvent` to the engine via RabbitMQ.

3.7.2 Worker Implementation

Workers execute assigned commands and report completion with energy usage. Power measurement is integrated using `pyRAPL`, enabling direct reporting of package-level energy consumption where supported.

Listing 3.2: Worker task execution with energy measurement (excerpt).

```
async def handle_start_task(self, command: StartTaskCommand)
   -> None:
    pyRAPL.setup()
    meter = pyRAPL.Measurement("power_consumption")
    meter.begin()
    process = await asyncio.create_subprocess_shell(
        command.task_definition.command,
        stdout=asyncio.subprocess.PIPE,
        stderr=asyncio.subprocess.PIPE,
    )
    stdout, _ = await process.communicate()
    meter.end()
    power_consumed = -1
    if meter.result.pkg is not None:
        power_consumed = meter.result.pkg[0]
    await self.emit(
        TaskCompletionEvent(
            task_id=command.task_id,
            energy_consumed=power_consumed,
            result=stdout.decode(),
        )
    )
```

It should be noted that to execute the code above, the worker process must have appropriate permissions to access the RAPL interface, which may require running with elevated privileges or configuring access to the relevant hardware counters.

In the experiements that were run in the same host, these values are mocked to match the execution length and the simulated host's power profile, since the actual energy consumption would not be meaningful in a simulated environment. This allows the scheduler to receive realistic energy feedback without needing actual power measurement hardware in the loop.

The worker runtime connects to the broker, consumes commands, and emits periodic updates as `WorkerUpdateEvent` with current properties (e.g., memory availability, core availability). This supports live engine decisions without direct coupling to worker processes.

Listing 3.3: Worker runtime event loop (excerpt).

```
async def run_worker(config: Worker):
    worker_id = WorkerId(value=config.worker_id)
    connector = RabbitMQConnector(url=config.rabbitmq_url)
    await connector.initialize()
    await connector.connect(worker_id)

    async def send_to_engine(event: EngineEvent) -> None:
        await connector.send_to_engine(event.model_dump_json()
            )

    worker = CoreWorker(emit_callback=send_to_engine,
       properties=config.properties)

    async def consume_from_engine(msg: str) -> None:
        command = EngineCommand.parse(msg)
        await worker.consume(command)

    async def send_updates_to_engine():
        while True:
            await asyncio.sleep(config.update_interval_seconds
                )
            event = WorkerUpdateEvent(worker_id=worker_id,
               properties=worker.properties)
            await send_to_engine(event)

    consume_task = asyncio.create_task(
        connector.consume_from_engine(worker_id,
           consume_from_engine)
    )
    update_task = asyncio.create_task(send_updates_to_engine()
       )
    registration_event = WorkerRegistrationEvent(
        worker_id=worker_id,
        properties=config.properties,
    )
    await connector.send_to_engine(registration_event.
       model_dump_json())
    await consume_task
    await update_task
```

3.7.3 Engine Implementation

The engine is the orchestration core that consumes worker events, maintains task and worker state, and emits start commands according to a pluggable scheduling strategy. Its primary responsibilities are:

- Parse incoming events and dispatch to handlers.
- Track per-task state transitions and dependency satisfaction.
- Invoke the scheduler at a controlled cadence.
- Issue `StartTaskCommand` to workers.

Listing 3.4: Engine event handling and scheduling trigger (excerpt).

```
async def consume(self, event: EngineEvent):
    """Dispatch incoming events to appropriate handlers and
       trigger scheduling when necessary."""

    if isinstance(event, WorkerRegistrationEvent):
        return await self.handle_worker_registration(event)
    if isinstance(event, WorkerUpdateEvent):
        return await self.handle_worker_update(event)
    if isinstance(event, TaskCompletionEvent):
        return await self.handle_task_completion(event)
    if isinstance(event, WorkflowSubmissionEvent):
        return await self.handle_workflow_submission(event)


async def handle_task_completion(self, event:
   TaskCompletionEvent):
    """Handle task completion event emitted by a worker"""

    # Update task state with completion time, energy consumed,
       and QoS penalty
    current_task = self.tasks[event.task_id]
    current_task.state.completed_at = time.time()
    current_task.state.energy_consumed = event.energy_consumed
    current_task.state.qos_penalty = (
        current_task.state.started_at - self.engine_start_time
    ) * current_task.definition.priority

    # Mark dependent tasks as schedulable and trigger
       scheduling
```

```
    self.mark_schedulable_tasks()
    await self.trigger_scheduling()
```

The core event loop includes handlers for worker registration, worker updates, workflow submission, and task completion. The engine also computes a QoS penalty at completion time and marks newly eligible tasks as schedulable.

Scheduling is rate-limited by a fixed interval to avoid excessive rescheduling. When a scheduler returns a valid `(worker_id, task_id)` pair, the engine validates compatibility, emits a `StartTaskCommand`, and updates task and worker state.

Listing 3.5: Scheduling decision emission and state updates (excerpt).

```
async def trigger_scheduling(self):
    """Trigger the scheduling strategy to select the next task
       -worker assignment."""

    # Rate limit scheduling to avoid thrashing
    if time.time() - self.last_scheduled_at <
       TIME_BETWEEN_SCHEDULING:
        return

    # Query the scheduler for the next assignment
    self.last_scheduled_at = time.time()
    scheduling_result = await self.scheduler.schedule(
        list(self.workers.values()),
        list(self.tasks.values()),
    )
    if scheduling_result is None:
        # No valid scheduling decision at this time
        return

    # Emit a StartTaskCommand to the selected worker
    worker_id, task_id = scheduling_result
    task = self.tasks[task_id]
    await self.emit(
        StartTaskCommand(
            worker_id=worker_id,
            task_id=task_id,
            task_definition=task.definition,
        )
    )

    # Update task state to reflect scheduling decision
    task.state.is_scheduled = True
```

```
    task.state.is_schedulable = False
    task.state.executed_worker_id = worker_id
    task.state.started_at = time.time()
```

Here, the `self.tasks` dictionary is updated with discovered tasks from workflow submissions, and the `self.workers` dictionary is updated with registered workers and their properties. The workers are expected to periodically emit updates with their current properties using the `WorkerUpdateEvent`, which the engine consumes to keep its view of worker states up to date for scheduling decisions.

The engine runtime wires the core to the RabbitMQ connector, selects a strategy based on CLI configuration, and continuously renders a Gantt chart for live visual inspection. After completion, it dumps task state to a JSON file to be used for post-run metrics analysis.

Currently, the engine supports four scheduling strategies: random, DRL-based, HEFT, and FERPTS. The DRL strategy uses the GNN agent described in the next section, while HEFT and FERPTS are heuristic algorithms implemented in the strategies module. The random strategy serves as a baseline for comparison.

Listing 3.6: Engine runtime wiring and task dump (excerpt).

```
async def run_engine(config: Engine, stop_when: Callable[[
   CoreEngine], bool] | None = None):
    """Run the engine with the specified configuration and
       scheduling strategy."""

    # Select the scheduling strategy based on configuration
    scheduler: SchedulerStrategy = RandomSchedulingStrategy()
    if config.algorithm == "drl":
        scheduler = DRLScheduler()
    if config.algorithm == "heft":
        scheduler = HeftSchedulingStrategy()
    if config.algorithm == "ferpts":
        scheduler = FerptsSchedulingStrategy()
    if config.algorithm == "rr":
        scheduler = RoundRobinSchedulingStrategy()

    # Initialize the message broker connector and the engine
       core
    connector = RabbitMQConnector(url=config.rabbitmq_url)
    await connector.initialize()
    engine = CoreEngine(emit_callback=send_to_worker,
       scheduler=scheduler)
    gantt_task = asyncio.create_task(run_gantt_updater(engine,
```

```
        output_path="gantt.png", interval=1.0))
    # ...
    with open("tasks_dump.json", "w") as f:
        json.dump([task.model_dump() for task in engine.tasks.
            values()], f, indent=2)
```

3.7.4 GNN Agent (2SD-GAT) for Scheduling

The strategies module includes a GNN agent used by the DRL scheduler. The implementation corresponds to the 2SD-GAT policy used for inference in the scheduler.

Listing 3.7: GAT-based task encoder and two-stage actor (excerpt).

```
class GnnTaskEncoder(nn.Module):
    """GAT encoder for task features and dependencies."""

    def __init__(self, hidden_dim: int, embedding_dim: int,
        device: torch.device) -> None:
        super().__init__()
        self.device = device
        self.embedding_dim = embedding_dim
        self.network = GAT(
            in_channels=F_TASK,
            hidden_channels=hidden_dim,
            num_layers=3,
            out_channels=embedding_dim,
            heads=4,
            concat=False,
        ).to(device)

    def forward(self, task_features: torch.Tensor,
        dependencies: torch.Tensor) -> tuple[torch.Tensor,
        torch.Tensor]:
        bsz = task_features.shape[0]
        task_features_flat = task_features.reshape(bsz *
            N_TASK, F_TASK)
        edge_indices: list[torch.Tensor] = []
        for i in range(bsz):
            edge_index = torch.nonzero(dependencies[i],
                as_tuple=False).T
            edge_indices.append(edge_index + N_TASK * i)
        edge_indices_flat = torch.cat(edge_indices, dim=1)
        task_encoding_flat: torch.Tensor = self.network(
```

```
            task_features_flat, edge_index=edge_indices_flat)
        task_encodings = task_encoding_flat.reshape(bsz,
            N_TASK, self.embedding_dim)
        task_pool = task_encodings.mean(dim=1)
        return task_encodings, task_pool


class GnnVmEncoder(nn.Module):
    """MLP encoder for VM features."""

    def __init__(self, hidden_dim: int, embedding_dim: int,
        device: torch.device) -> None:
        super().__init__()
        self.device = device
        self.embedding_dim = embedding_dim
        self.network = nn.Sequential(
            nn.Linear(F_VM, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, embedding_dim),
        ).to(device)

    def forward(self, vm_features: torch.Tensor) -> tuple[
        torch.Tensor, torch.Tensor]:
        bsz = vm_features.shape[0]
        vm_features_flat = vm_features.reshape(bsz * N_VM,
            F_VM)
        vm_encoding_flat: torch.Tensor = self.network(
            vm_features_flat)
        vm_encoding = vm_encoding_flat.reshape(bsz, N_VM, self
            .embedding_dim)
        vm_pool = vm_encoding.mean(dim=1)
        return vm_encoding, vm_pool


class GnnDecoder(nn.Module):
    """MLP decoder for task or VM selection based on combined
        encodings."""
```

```
    def __init__(self, hidden_dim: int, embedding_dim: int,
        device: torch.device) -> None:
        super().__init__()
        self.device = device
        self.network = nn.Sequential(
            nn.Linear(3 * embedding_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, 1),
        ).to(device)

    def forward(
        self, encoding: torch.Tensor, mask: torch.Tensor,
            task_pool: torch.Tensor, vm_pool: torch.Tensor
    ) -> torch.Tensor:
        bsz = encoding.shape[0]
        num_x = encoding.shape[1]
        encoding_flat = encoding.reshape(bsz * num_x, -1)
        rep_task_pool = task_pool.repeat(num_x, 1)
        rep_vm_pool = vm_pool.repeat(num_x, 1)
        comb_encoding = torch.cat([encoding_flat,
            rep_task_pool, rep_vm_pool], dim=1)
        scores_flat: torch.Tensor = self.network(comb_encoding
            )
        scores = scores_flat.reshape(bsz, num_x)
        scores.masked_fill_(mask == 0, -1e8)
        return scores


class GnnAgentActor(nn.Module):
    """Two-stage actor that first selects a task and then
       selects a VM based on the selected task's encoding."""

    def __init__(self, device: torch.device, embedding_dim:
        int = 32, hidden_dim: int = 64):
        super().__init__()
        self.device = device
        self.task_encoder = GnnTaskEncoder(hidden_dim=
            hidden_dim, embedding_dim=embedding_dim, device=
```

```
            device)
        self.vm_encoder = GnnVmEncoder(hidden_dim=hidden_dim,
            embedding_dim=embedding_dim, device=device)
        self.task_decoder = GnnDecoder(hidden_dim=hidden_dim,
            embedding_dim=embedding_dim, device=device)
        self.vm_decoder = GnnDecoder(hidden_dim=hidden_dim,
            embedding_dim=embedding_dim, device=device)

    def forward(
        self, x: torch.Tensor, action: torch.Tensor | None =
            None
    ) -> tuple[torch.Tensor, torch.Tensor, torch.Tensor]:
        decoded_obs = _decode_env_obs_batched(x.to(self.device
            ))
        bsz = decoded_obs.vm_features.shape[0]

        task_features = decoded_obs.task_features
        vm_features = decoded_obs.vm_features
        dependencies = decoded_obs.task_dependencies
        task_encoding, task_pool = self.task_encoder(
            task_features, dependencies)

        avg_vm_features = vm_features.mean(dim=1)
        _, avg_vm_pool = self.vm_encoder(avg_vm_features)

        task_mask = decoded_obs.task_mask
        task_logits: torch.Tensor = self.task_decoder(
            task_encoding, task_mask, task_pool, avg_vm_pool)
        task_probs = torch.softmax(task_logits, dim=1)
        task_dist = torch.distributions.Categorical(task_probs
            )
        chosen_task = task_dist.sample() if action is None
            else action // N_VM
        task_log_prob = task_dist.log_prob(chosen_task)
        task_entropy = task_dist.entropy()

        chosen_vm_features = vm_features[torch.arange(bsz),
            chosen_task]
        vm_encoding, vm_pool = self.vm_encoder(
            chosen_vm_features)

        chosen_vm_mask = decoded_obs.vm_mask[torch.arange(bsz)
```

```
            , chosen_task]
        vm_logits: torch.Tensor = self.vm_decoder(vm_encoding,
            chosen_vm_mask, task_pool, vm_pool)
        vm_probs = torch.softmax(vm_logits, dim=1)
        vm_dist = torch.distributions.Categorical(vm_probs)
        chosen_vm = vm_dist.sample() if action is None else
            action % N_VM
        vm_log_prob = vm_dist.log_prob(chosen_vm)
        vm_entropy = vm_dist.entropy()

        chosen_action = chosen_task * N_VM + chosen_vm
        total_log_prob = task_log_prob + vm_log_prob
        total_entropy = task_entropy + vm_entropy
        return chosen_action, total_log_prob, total_entropy


class GnnAgentCritic(nn.Module):
    """Critic that estimates the state value based on combined
        task and VM encodings."""

    def __init__(self, hidden_dim: int, embedding_dim: int,
        device: torch.device) -> None:
        super().__init__()
        self.device = device
        self.task_encoder = GnnTaskEncoder(hidden_dim=
            hidden_dim, embedding_dim=embedding_dim, device=
            device)
        self.vm_encoder = GnnVmEncoder(hidden_dim=hidden_dim,
            embedding_dim=embedding_dim, device=device)
        self.network = nn.Sequential(
            nn.Linear(2 * embedding_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, 1),
        ).to(device)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        decoded_obs = _decode_env_obs_batched(x.to(self.device
            ))
```

```
        task_features = decoded_obs.task_features
        dependencies = decoded_obs.task_dependencies
        _, task_pool = self.task_encoder(task_features,
            dependencies)
        avg_vm_features = decoded_obs.vm_features.mean(dim=1)
        _, avg_vm_pool = self.vm_encoder(avg_vm_features)
        comb_encoding = torch.cat([task_pool, avg_vm_pool],
            dim=1)
        state_value: torch.Tensor = self.network(comb_encoding
            )
        return state_value.squeeze(dim=-1)


class GnnAgent(nn.Module):
    """GNN-based agent that combines the actor and critic for
       DRL scheduling."""

    def __init__(self, device: torch.device, embedding_dim:
       int = 8, hidden_dim: int = 64):
        super().__init__()
        self.device = device
        self.actor = GnnAgentActor(hidden_dim=hidden_dim,
            embedding_dim=embedding_dim, device=device)
        self.critic = GnnAgentCritic(hidden_dim=hidden_dim,
            embedding_dim=embedding_dim, device=device)

    def get_action_and_value(
        self, x: torch.Tensor, action: torch.Tensor | None =
            None
    ) -> tuple[torch.Tensor, torch.Tensor, torch.Tensor, torch
       .Tensor]:
        chosen_action, log_prob, entropy = self.actor(x,
            action)
        value: torch.Tensor = self.critic(x)
        return chosen_action, log_prob, entropy, value


def _decode_env_obs_batched(tensor: torch.Tensor) ->
   EnvObsTensor:
    """Decode a batch of environment observation tensors into
       structured components."""
```

```
    bsz = tensor.shape[0]
    assert tensor.shape == (bsz, OBS_SIZE), tensor.shape

    task_features_list: list[torch.Tensor] = []
    vm_features_list: list[torch.Tensor] = []
    task_mask_list: list[torch.Tensor] = []
    vm_mask_list: list[torch.Tensor] = []
    task_dependencies_list: list[torch.Tensor] = []
    for tensor_i in tensor:
        task_features_list.append(tensor_i[: N_TASK * F_TASK].
           reshape(N_TASK, F_TASK))
        offset = N_TASK * F_TASK
        vm_features_list.append(tensor_i[offset : offset +
           N_TASK * N_VM * F_VM].reshape(N_TASK, N_VM, F_VM))
        offset += N_TASK * N_VM * F_VM
        task_mask_list.append(tensor_i[offset : offset +
           N_TASK].long())
        offset += N_TASK
        vm_mask_list.append(tensor_i[offset : offset + N_TASK
           * N_VM].reshape(N_TASK, N_VM).long())
        offset += N_TASK * N_VM
        task_dependencies_list.append(tensor_i[offset : offset
            + N_TASK * N_TASK].reshape(N_TASK, N_TASK).long())

    return EnvObsTensor(
        task_features=torch.stack(task_features_list),
        vm_features=torch.stack(vm_features_list),
        task_mask=torch.stack(task_mask_list),
        vm_mask=torch.stack(vm_mask_list),
        task_dependencies=torch.stack(task_dependencies_list),
    )
```

The DRL strategy constructs a fixed-size observation tensor, loads a trained model checkpoint, and maps the agent's chosen action to a valid `(worker_id, task_id)` pair before issuing a start command.

Listing 3.8: DRL strategy using the GNN agent (excerpt).

```
def _encode_env_obs(obs: EnvObs) -> np.ndarray:
    """Encode the environment observation into a fixed-size
       numpy array for input to the GNN agent."""

    arr = np.concatenate(
        [
```

```
            np.asarray(obs.task_features, dtype=np.float32).
               flatten(),
            np.asarray(obs.vm_features, dtype=np.float32).
               flatten(),
            np.asarray(obs.task_mask, dtype=np.int32),
            np.asarray(obs.vm_mask, dtype=np.int32).flatten(),
            np.asarray(obs.task_dependencies, dtype=np.int32).
               flatten(),
        ]
    )
    assert arr.shape == (OBS_SIZE,), arr.shape
    return arr


class DRLScheduler(SchedulerStrategy):
    """RL-based scheduling strategy that uses a GNN agent to
       select task-worker assignments based on the current
       system state."""

    async def schedule(self, workers: list[WorkerState], tasks
       : list[Task]):
        """Schedule the next task-worker assignment based on
           the current state of workers and tasks."""

        # Buil and encode the environment observation for the
           GNN agent
        obs = _build_obs(workers, tasks)
        encoded_obs = _encode_env_obs(obs)

        # Load the trained GNN agent model checkpoint
        device = torch.device("cuda" if torch.cuda.
           is_available() else "cpu")
        agent = GnnAgent(device=device)
        state = torch.load(MODEL_PATH, weights_only=True,
           map_location=device)
        agent.load_state_dict(state)
        agent.eval()

        # Use the agent to select an action based on the
           encoded observation
        with torch.no_grad():
            encoded_obs_tensor = torch.tensor(encoded_obs,
```

```
                dtype=torch.float32, device=agent.device).
                unsqueeze(0)
            action, _, _, _ = agent.get_action_and_value(
                encoded_obs_tensor)
        action_idx = int(action.item())
        task_idx = action_idx // N_VM
        worker_idx = action_idx % N_VM

        # Map the selected action index to a valid (worker_id,
            task_id) pair and validate the scheduling decision
        worker_id, task_id = _map_indices_to_worker_and_task(
            workers, tasks, worker_idx, task_idx)
        _validate_scheduling_decision(worker_id, task_id,
            workers, tasks)
        return worker_id, task_id
```

3.7.5 Web GUI for Workflow Simulation

The above tools were initially designed for CLI-based interaction, which is suitable for scripted experiments and batch runs. However, to facilitate interactive experimentation and visualization of scheduling behavior, a lightweight web GUI was developed as an auxiliary tool. This GUI, implemented as a web application, uses Python's `ThreadingHTTPServer` to serve static assets expose a small REST API for interacting with the engine and workers. This tool can also be found in the same public repository as the other components (https://github.com/kdsuneraavinash/prototype-workflow-engine in the `admin` directory).

The GUI focuses on end-to-end workflow simulation. When simulating, users can configure hosts, VMs, workflows, and tasks through editable tables, adjust objective weights for makespan, energy, and SLA penalties, and export or import JSON payloads. It is possible to run the engine only and connect external workers, or run both the engine and workers directly from the interface, in which case it will start workers in the local environment based on the VM definitions provided in the tables.

The interface provides three synchronized views:

- A engine management panel for configuring the scheduler and starting/stopping the engine process. The engine will be started on the same host as the GUI, and will connect to the shared RabbitMQ instance.

- A setup visualizer that renders hosts, VMs, and DAGs from the current table state (including dependency edges). The host and VM information is only used for visualization and simulation purposes, and does not affect the actual scheduling

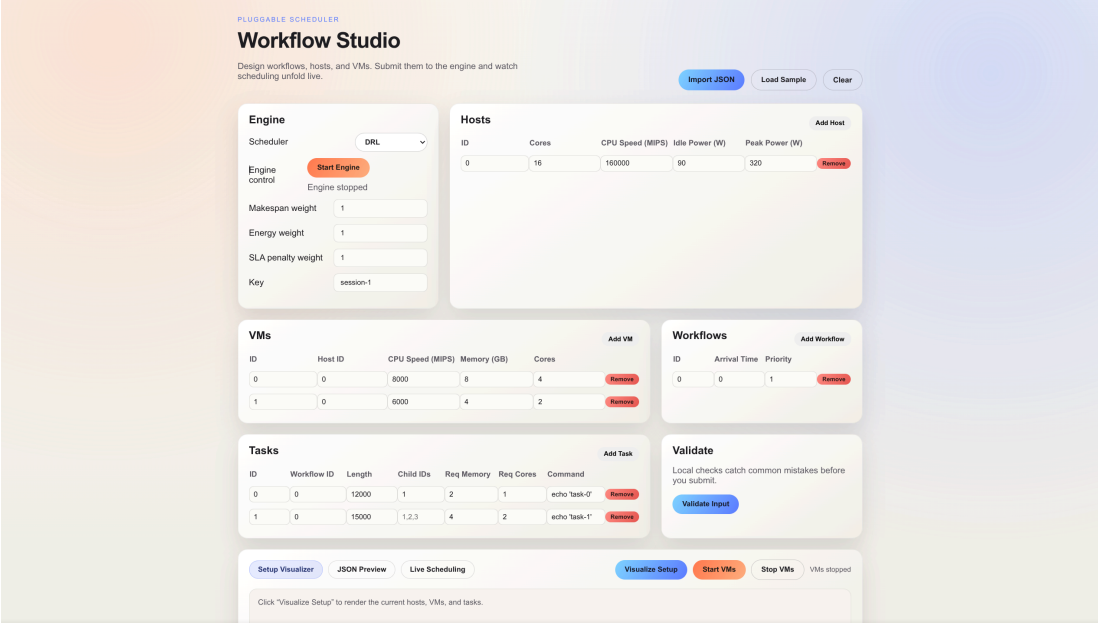

Fig. 3.24: Screenshot of the web GUI for workflow simulation and scheduling visualization

decisions of the engine, which are based on the task definitions and worker updates received through RabbitMQ (FigureB.4).

- A live scheduling tab that refreshes a generated Gantt chart image on a short interval. This allows users to observe the scheduling decisions made by the engine in real time as the workflow executes.

This GUI provides a repeatable workflow for configuring simulations, starting the engine and workers, and observing schedule outputs without leaving the browser, while still using the same engine and messaging pipeline.

CHAPTER 4

RESULTS AND ANALYSIS

4.1 Results and Analysis for GNN-Flat

4.1.1 Results and Discussion

The performance of the proposed GNN-Flat framework was evaluated across three datasets (DS_1, DS_2, and DS_3), each representing varying levels of complexity and task concurrency. The results demonstrate consistent improvements in makespan and energy efficiency compared to baseline methods based on heuristics. The results and a detailed discussion of the findings are presented below.

The results of the proposed algorithm are compared with the baseline methods described in Table 3.6. The evaluation metrics include makespan, active energy consumption, Energy-Delay Product (EDP), and Energy Per Task (EPT). Table 4.1 summarizes the performance of the proposed algorithm and baseline methods across these metrics. The table contains the average of 10 samples taken from the dataset parameters shown in Table 3.5.

In Dataset DS_1, which represents a low-concurrency environment with relatively small workflows, the proposed framework achieved a makespan of 194.71 s, surpassing the closest heuristic, HEFT, which recorded 201.87 s. This corresponds to a reduction of approximately 3.55% in makespan. In terms of active energy consumption, the proposed method achieved the lowest value of 3111.63 J, marginally outperforming Max-Min, which recorded 3125.44 J. These results userlines the framework's ability to provide balanced optimization in environments characterized by smaller workflows and limited task interdependencies.

For Dataset DS_2, which represents a moderately complex environment with larger workflows and increased variability, the proposed method demonstrated significant improvements in makespan and energy efficiency. It achieved a makespan of 689.22 s, which is a reduction of 13.92% compared to Round Robin, the closest competitor (800.72 s). Additionally, the framework recorded the lowest active energy consumption of 10964.45 J, slightly better than Min-Min (10998.52 J). These findings highlight the proposed framework's effectiveness in handling moderately complex scheduling scenarios by leveraging its ability to model task dependencies and resource constraints dynamically.

In Dataset DS_3, characterized by a similar environment to DS_2, but with a higher number of tasks inside a workflow (15-20 in DS_2 vs 40-50 in DS_3), resulting in higher computational demands, the proposed framework exhibited robust performance. It achieved the lowest makespan of 1779.61 s, surpassing HEFT (1837.80 s) with

TABLE 4.1: PERFORMANCE COMPARISON OF THE PROPOSED ALGORITHM AND BASELINES

Dataset	**Algorithm**	**Makespan (s)**	**Active Energy Cons. (J)**	**EDP (Js)** $E_{\text{total}} \times M$	**EPT (J)** $\frac{E_{\text{active}}}{N^{\text{Tasks}}}$	**Decision Latency (s)**	**Total Run Time (s)**
DS_1	Random	306.61	3225.12	130782928.46	103.39	0.01	0.01
	Round Robin	246.39	3140.78	105123167.65	100.65	0.01	0.01
	Max-Min	210.27	3125.44	89671938.18	100.26	0.01	0.01
	Min-Min	206.54	3128.09	88041738.35	100.38	0.01	0.01
	HEFT	201.87	3138.89	85887677.23	100.59	0.01	0.01
	Proposed	**194.71**	**3111.63**	**82751239.30**	**99.80**	0.01	0.19
DS_2	Random	1009.21	11027.32	962195838.37	63.15	0.01	0.01
	Round Robin	800.72	11525.60	764351107.50	66.02	0.01	0.01
	Max-Min	924.37	11028.04	880585573.80	63.15	0.01	0.01
	Min-Min	876.27	10998.52	836079990.97	63.00	0.01	0.01
	HEFT	876.81	11173.97	835583187.95	63.99	0.08	0.08
	Proposed	**689.22**	**10964.45**	**657333112.54**	**62.79**	0.01	2.50
DS_3	Random	2494.47	29127.95	5294475677.89	64.20	0.01	0.01
	Round Robin	2056.44	30536.36	4366303181.27	67.30	0.01	0.01
	Max-Min	2272.08	28939.87	4819828319.41	63.79	0.01	0.01
	Min-Min	2170.36	28720.47	4604694515.36	63.32	0.01	0.01
	HEFT	1837.80	29178.38	3899255596.67	64.32	0.36	0.36
	Proposed	**1779.61**	**28611.36**	**3774081849.42**	**63.08**	0.05	21.32
DS_3^L	Random	3051.34	36554.21	7893707255.36	81.36	0.02	0.02
	Round Robin	2539.95	38456.79	6575378599.92	85.60	0.02	0.02
	Max-Min	2808.50	35872.85	7266347878.71	79.85	0.02	0.02
	Min-Min	2685.12	36370.58	6948309882.74	80.96	0.02	0.02
	HEFT	2293.07	36842.30	5932778716.71	82.01	0.34	0.34
	Proposed	**2238.99**	**35686.71**	**5792398179.82**	**79.43**	0.04	20.12
DS_3^R	Random	1883.77	21241.13	3071269569.34	47.49	0.03	0.03
	Round Robin	1548.01	22274.42	2526040425.56	49.80	0.03	0.03
	Max-Min	1704.66	21146.16	2779509096.67	47.27	0.03	0.03
	Min-Min	1683.71	20981.35	2745459751.10	46.91	0.03	0.03
	HEFT	1421.96	21444.84	2318718160.47	47.95	0.41	0.41
	Proposed	**1330.26**	**20864.43**	**2169170282.82**	**46.65**	0.05	23.85

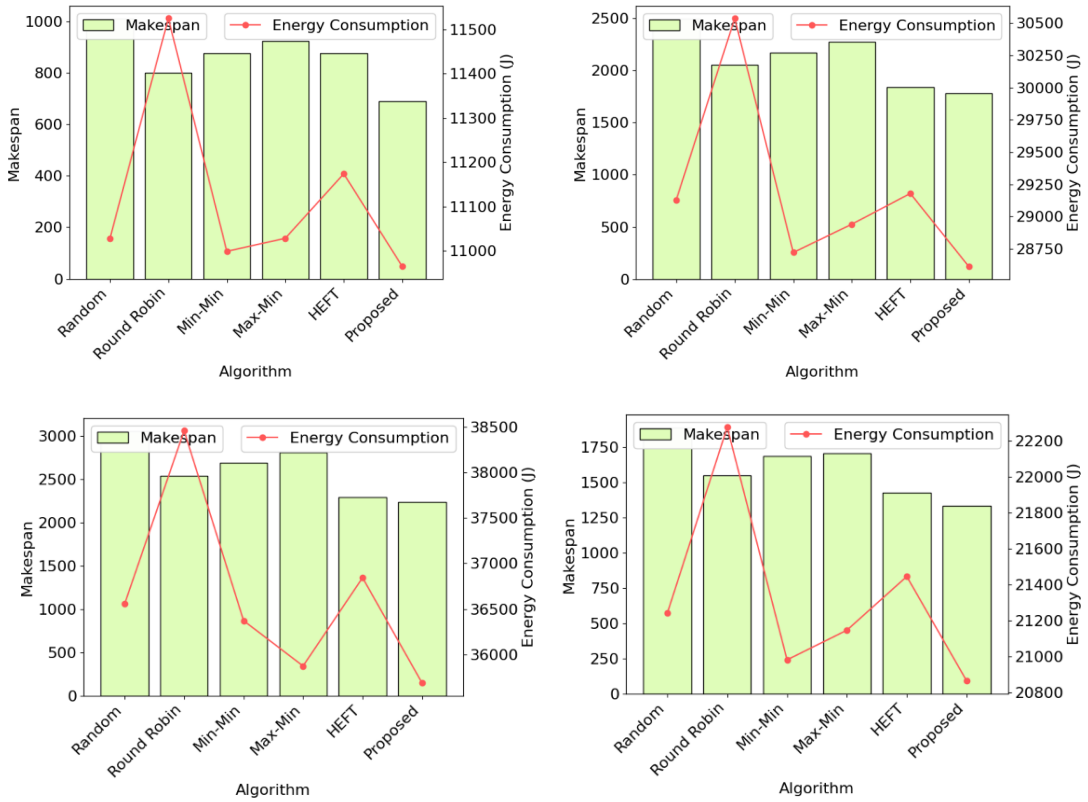

Fig. 4.1: Comparison of Makespan and Energy Consumption Across Algorithms

Note. DS_2 (top left), DS_3 (top right), DS_3^L (bottom left), and DS_3^R (bottom right)

an improvement of 3.17%. The proposed method recorded a slightly lower energy consumption (28611.36 J) compared to Min-Min (28720.47 J). The result of achieving a lower makespan while maintaining a lower energy-consumption demonstrates that the framework maintains competitive energy efficiency even in highly demanding scenarios. This result underscores the scalability and adaptability of the proposed method in dynamic and complex cloud environments.

In Dataset DS_3^L, with a left-skewed distribution favoring shorter tasks, and DS_3^R, with a right-skewed distribution emphasizing longer tasks, the proposed framework consistently demonstrated superior performance across both scenarios. For DS_3^L, the framework achieved the lowest makespan of 2238.99 s, outperforming HEFT (2238.99 s) by 2.36%, and recorded the lowest active energy consumption of 35,686.71 J, marginally surpassing Max-Min (35,872.85 J). In DS_3^R, the proposed method excelled further with a makespan of 1330.26 s, a 6.45% improvement over HEFT (1421.96 s), and the lowest energy consumption of 20,864.43 J, narrowly outperforming Min-Min (20,981.35 J). These results underscore the framework's adaptability, effectively optimizing makespan and energy consumption across diverse workload distributions in dynamic cloud environments with varying workloads.

To complement the discussion of results, EDP and EPT metrics provide deeper insights into the trade-offs between makespan reduction and energy efficiency. The EDP, calculated as the product of active energy consumption and makespan, highlights the cumulative energy cost associated with achieving faster task completion times. A lower EDP indicates a better balance between time and energy efficiency. Similarly, the EPT metric normalizes energy consumption over the number of tasks, offering a task-level perspective on energy efficiency. Across datasets, the proposed method consistently achieves the lowest EDP and EPT values, demonstrating its ability to optimize energy consumption while maintaining competitive makespan performance. These metrics reinforce the multi-objective effectiveness of the proposed algorithm.

4.1.2 Algorithm Runtime Analysis

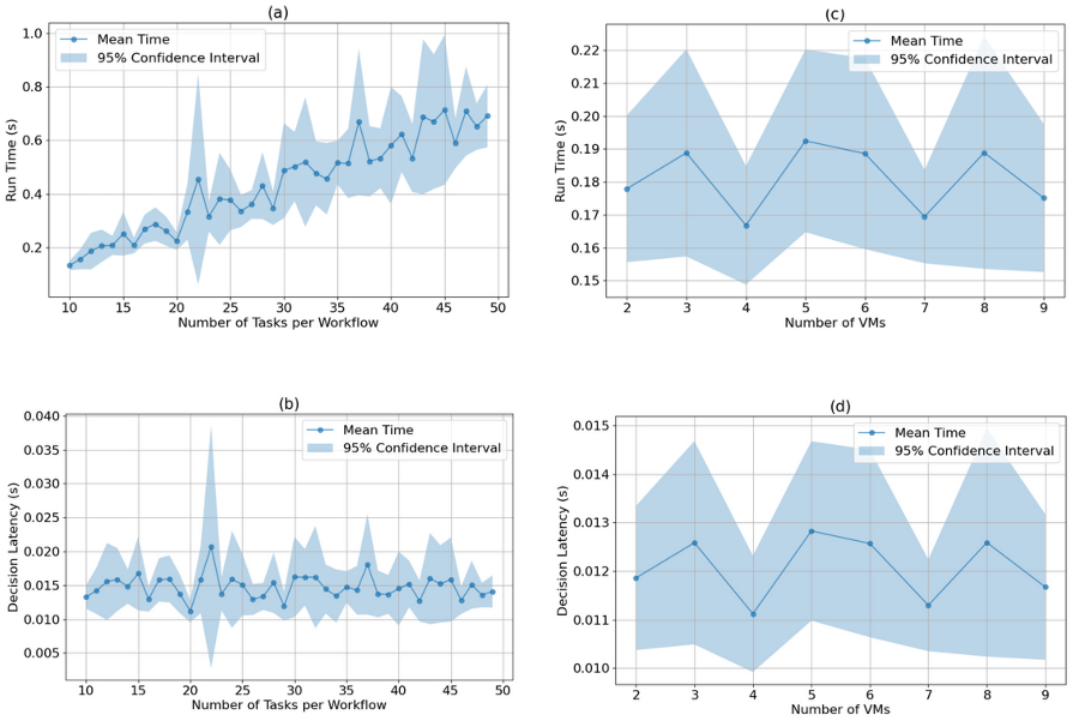

Fig. 4.2: Runtime and Decision latency variation with different parameters

The runtime of the proposed algorithm is slightly higher compared to heuristic-based approaches; however, it does not exhibit exponential growth. Unlike batch-based heuristic algorithms that compute scheduling solutions in a single step, the proposed approach operates as an online scheduling algorithm, making scheduling decisions iteratively. Consequently, while the total runtime is relatively high, the per-decision latency remains low. Since each decision in the scheduling process is made via inference using a trained reinforcement learning model, the computational cost per decision

remains constant, primarily determined by the number of parameters in the model. This ensures scalability even as task complexity grows.

To further analyze the performance, runtime of the algorithm across various scenarios was plotted as per figure 4.2. As the number of tasks per workflow increases, the total runtime increases in a linear fashion as shown in figure 4.2 (a) and (b), while the decision latency remains somewhat constant. As the number of VMs increases, the runtime does not change drastically as indicated in figure 4.2 (c) and (d). This behavior is explained by the fact that number of tasks influence the total number of decisions the algorithm must make, which in turn affects total runtime. However, the time required per decision remains low, making the proposed approach suitable for real-time and dynamic scheduling scenarios.

4.1.3 Statistical Analysis

Table 4.2 provides the results of statistical tests conducted to evaluate the performance of different scheduling algorithms across three datasets (DS_1, DS_2, and DS_3) for two key metrics: Makespan and Energy. The table includes the results of the Friedman test, a non-parametric statistical test used to detect overall differences among the algorithms, followed by Wilcoxon signed-rank tests, which perform pairwise comparisons between the proposed algorithm and other baseline algorithms to determine if the observed differences are statistically significant.

For Makespan, the Friedman test indicates statistically significant differences among the algorithms across all datasets (p-value = 0.0000). The pairwise Wilcoxon tests show that the proposed algorithm consistently outperforms all other methods, with statistically significant improvements (p-value < 0.05) against Random, Round Robin, Min-Min, and Max-Min across all datasets. The results for HEFT, while better than other baselines in some cases, show non-significant differences (p-value > 0.05) for DS_1 and DS_3, indicating comparable performance in certain scenarios. For Energy, the Friedman test results vary across datasets. For DS_1, no statistically significant overall differences are detected among the algorithms (p-value = 0.0619). However, for DS_2 and DS_3, significant differences are observed (p-value < 0.05). Pairwise Wilcoxon tests highlight significant energy efficiency improvements for the proposed algorithm compared to Random and Round Robin methods across DS_2 and DS_3. Against Min-Min, Max-Min, and HEFT, the differences are less pronounced, with several non-significant results, particularly in DS_1. These findings suggest that while the proposed method improves energy efficiency, the magnitude of the improvement depends on the complexity and scale of the dataset.

TABLE 4.2: STATISTICAL TEST RESULTS FOR SCHEDULING ALGORITHMS ACROSS DATASETS

Metric	**Dataset**	**Friedman**	**p-value**	**Baseline**	**Wilcoxon**	**p-value**
Makespan	DS_1	38.00	0.0000	Random	0.00	0.0020
				Round Robin	0.00	0.0020
				Min-Min	7.00	0.0371
				Max-Min	7.00	0.0371
				HEFT	15.00	0.2324
	DS_2	41.03	0.0000	Random	0.00	0.0020
				Round Robin	0.00	0.0020
				Min-Min	0.00	0.0020
				Max-Min	0.00	0.0020
				HEFT	0.00	0.0020
	DS_3	44.63	0.0000	Random	0.00	0.0020
				Round Robin	0.00	0.0020
				Min-Min	0.00	0.0020
				Max-Min	0.00	0.0020
				HEFT	10.00	0.0840
Energy	DS_1	10.51	0.0619	Random	3.00	0.0098
				Round Robin	11.00	0.1055
				Min-Min	19.00	0.4316
				Max-Min	23.00	0.6953
				HEFT	22.00	0.6250
	DS_2	25.14	0.0001	Random	26.00	0.9219
				Round Robin	0.00	0.0020
				Min-Min	25.00	0.8457
				Max-Min	21.00	0.5566
				HEFT	3.00	0.0098
	DS_3	26.06	0.0001	Random	3.00	0.0098
				Round Robin	0.00	0.0020
				Min-Min	23.00	0.6953
				Max-Min	14.00	0.1934
				HEFT	1.00	0.0039

Note. Friedman statistics and p-values indicate overall differences among algorithms, while Wilcoxon statistics and p-values show pairwise comparisons between the proposed method and each baseline.

4.1.4 Multi-objective solution Analysis

To evaluate the multi-objective optimization capabilities of the proposed GNN-Flat framework, a Pareto front diagram was generated and is presented in Figure 4.3. The

Pareto front illustrates the trade-offs between makespan and active energy consumption for the proposed method and baseline algorithms. The proposed GNN-Flat framework achieves a superior trade-off. In comparison, baseline methods such as HEFT, Min-Min, Max-Min, Random, and Round-Robin are scattered further from the Pareto front, reflecting their limitations in optimizing both objectives simultaneously. While HEFT and Min-Min provide relatively competitive solutions, their points remain dominated by the proposed method, which consistently outperforms them in both makespan and energy consumption. These findings highlight the ability of the proposed framework to deliver dependency-aware, dynamic scheduling decisions while effectively balancing multiple objectives.

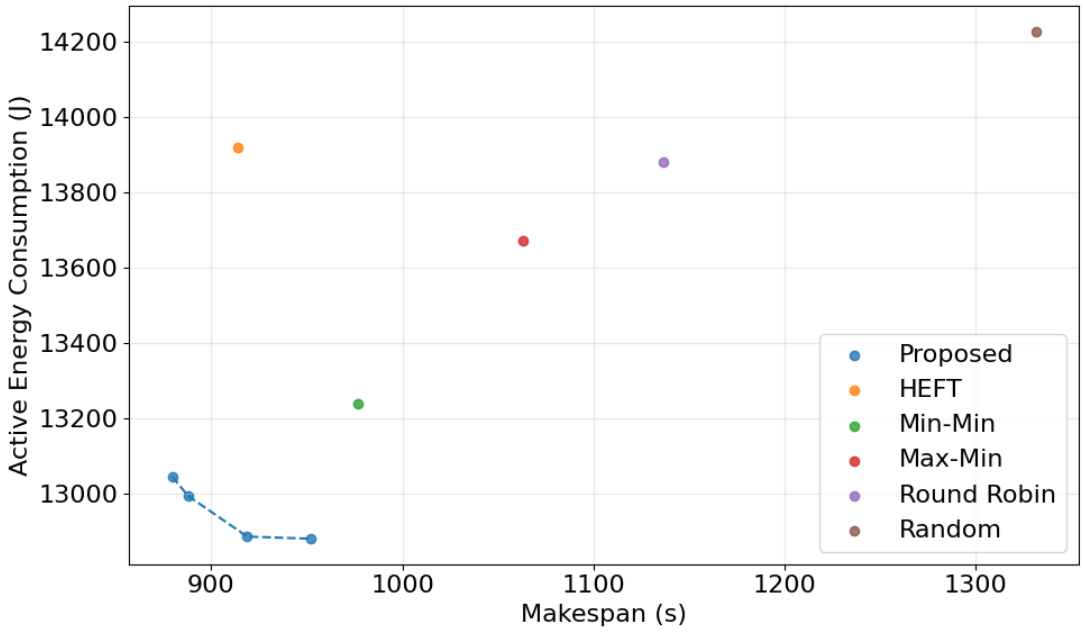

Fig. 4.3: Pareto front illustrating the trade-offs between makespan and active energy consumption achieved by the proposed GNN-Flat framework compared to baseline algorithms

When compared against a multi-objective scheduling algorithm such as MOHEFT [20], the performance of the proposed GNN-Flat framework was analyzed using two widely recognized multi-objective evaluation metrics, hypervolume and IGD. The hypervolume metric measures the volume in the objective space that is dominated by the obtained Pareto front relative to a predefined reference point, with a higher hypervolume indicating a better approximation of the true Pareto front[140]. IGD, on the other hand, quantifies the distance between the obtained Pareto front and a reference set of optimal trade-off solutions, with lower IGD values indicating better diversity and proximity to the optimal front[140].

The results indicate that while the proposed method achieves better makespan reduction, its obtained solutions exhibit a less diverse distribution compared to MOHEFT, as

evidenced by the lower hypervolume and higher IGD values (Table 4.3 and Figure 4.4). This suggests that the model, despite optimizing makespan and energy consumption simultaneously, lacks the flexibility to generate a broad range of trade-off solutions. The reduced diversity can be attributed to the fixed nature of the reward function, which does not allow for dynamically adjusting the importance of makespan versus energy consumption during scheduling. As a result, the learned policy tends to converge towards a specific balance, rather than exploring a wider range of potential Pareto-optimal solutions.

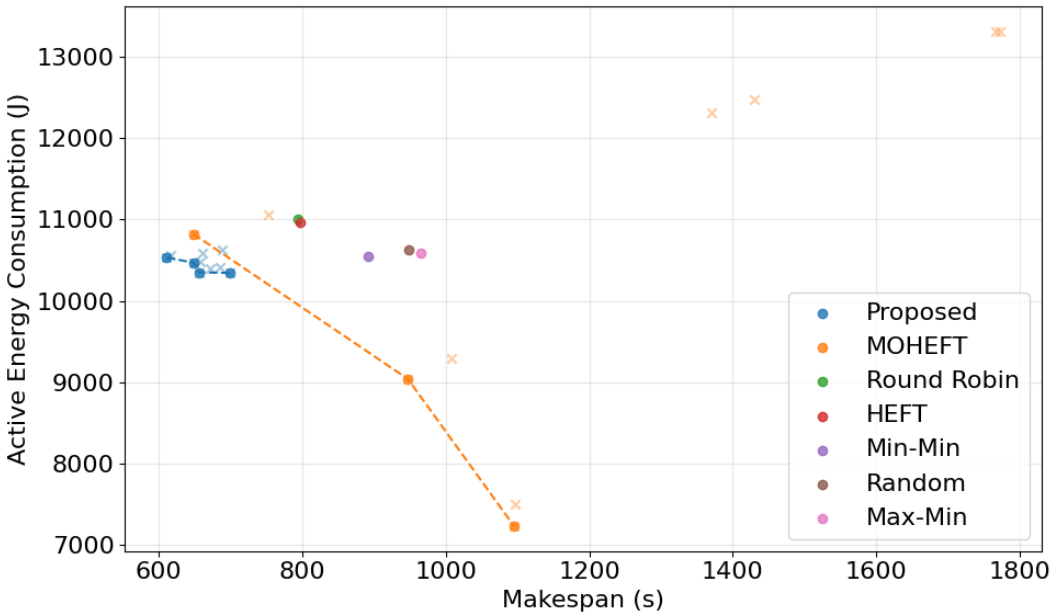

Fig. 4.4: Pareto front illustrating the trade-offs between makespan and active energy consumption achieved by the proposed GNN-Flat framework compared to baseline algorithms and MOHEFT algorithm

TABLE 4.3: PERFORMANCE AND SOLUTION COMPARISON OF THE PROPOSED ALGORITHM AND MOHEFT

Dataset	**Algorithm**	**Makespan (s)**	**Active Energy Cons. (J)**	**Hypervolume**	**IGD**
DS_1	MOHEFT	428.35	**2758.44**	1274777.84	2458.88
	Proposed	**194.71**	3111.63	983181.54	2869.48
DS_2	MOHEFT	1234.29	**10096.99**	7748022.07	7486.76
	Proposed	**689.22**	10964.45	4804159.93	10928.23
DS_3	MOHEFT	3245.56	**26123.24**	7748022.07	18594.15
	Proposed	**1779.61**	28611.36	4804159.93	27340.00

4.1.5 Limitations and Required Improvements

This study presents several notable contributions over existing work in cloud workflow scheduling. Firstly, the proposed approach supports DAG-based workflows, where task dependencies are effectively captured using GNNs that encode both task and VM states. This structural modeling enables the agent to learn context-aware scheduling policies based on the workflow topology and resource availability.

Secondly, the reward design departs from the conventional use of terminal rewards by employing an immediate reward structure, grounded in lower-bound heuristics for makespan and energy consumption. This approach accelerates learning and encourages the agent to make optimal decisions throughout the scheduling process, not just at the final timestep.

Thirdly, the framework demonstrates adaptability to dynamic environments. It is trained on a fixed configuration of hosts and VMs, yet generalizes effectively to different cluster sizes and task distributions. This generalization capability is enabled through the reinforcement learning loop, which continuously samples from the observation space, including VM characteristics and workflow structure, at every decision point. Unlike static heuristics, the RL-based scheduler dynamically adjusts to resource variations reflected in the environment. However, this implementation requires a restart to detect changes in VM or host availability. Enhancing the system to support real-time reconfiguration detection would further improve its practical applicability in dynamic cloud environments.

While the framework achieves consistent improvements in makespan and moderate gains in energy efficiency, the latter are not uniformly significant across all scenarios. This suggests that further refinement of the reward function may be necessary to more strongly incentivize energy-aware decisions. Moreover, although the inference time per decision remains largely stable, the computational demands of training pose scalability concerns, particularly for large-scale infrastructure or multi-tenant cloud systems.

As described in Section 3.2.6, the evaluation setup captures essential heterogeneity in CPU speeds, RAM allocations, and task characteristics. However, it abstracts uncertainties aspects and dynamic nature of real-world cloud environments.

Although the framework jointly optimizes makespan and energy consumption, it currently lacks the flexibility to produce diverse trade-off solutions. This is due to the static nature of the reward function, which does not allow the agent to adapt its prioritization based on runtime constraints or user preferences. A potential enhancement is to introduce an adjustable weighting mechanism that accepts objective weights as inputs. Such a mechanism would allow for dynamic balancing between objectives and generate a range of Pareto-optimal solutions tailored to user-defined priorities.

Several avenues for further research were identified in this section. These have been addressed in the 2SD-GAT, which incorporates two-stage cloud workflow scheduling:

- Introduced energy-specific reward shaping strategies that prioritize VM selection based on power efficiency.
- Incorporated additional objectives such as latency sensitivity to extend beyond performance and energy metrics.
- Implemented a two-stage agent architecture, where the first agent selects tasks and the second agent assigns them to VMs.
- Evaluated the framework using real-world-inspired workflow traces to assess its generalization and transferability.
- Benchmarked against a broader set of state-of-the-art multi-objective for comparative analysis.

4.2 Results and Analysis for 2SD-GAT

4.2.1 Results for Synthetic Datasets

This section presents a comprehensive evaluation of the proposed scheduling algorithm against a range of baseline methods under synthetic workload conditions. Table 4.4 and Table 4.5 reports the results for three synthetic datasets of increasing complexity. Corresponding visualizations are provided in Figures 4.5 and 4.6, which display the Pareto fronts and multi-objective metric comparisons.

Small-scale Workload

As shown in Figure 4.5, the Pareto front of 2SD-GAT overlaps significantly with several baselines. Given the relatively small number of tasks (50), the scheduling complexity is modest, allowing most algorithms to perform reasonably well. Nonetheless, 2SD-GAT is able to match or exceed the performance of both classical heuristics and multi-objective evolutionary algorithms.

From the Figure 4.6, Table 4.4, and Table 4.5, it can be observed that 2SD-GAT achieves the highest hypervolume value ($(4.958 \pm 0.040) \times 10^{10}$), indicating that its Pareto front spans a larger and more optimal portion of the objective space. GAT-Flat exhibits high variance in its results, likely because it was originally specialized for optimizing energy consumption and makespan; extending it to include QoS may have reduced its training effectiveness under the same iteration budget as 2SD-GAT. Although NSGA-III achieves a slightly better IGD (164.818), its decision latency is significantly higher (1.060 s) compared to 2SD-GAT (0.070 s). This trade-off highlights the real-time efficiency of 2SD-GAT: it achieves near-optimal Pareto performance at a fraction of the computational cost.

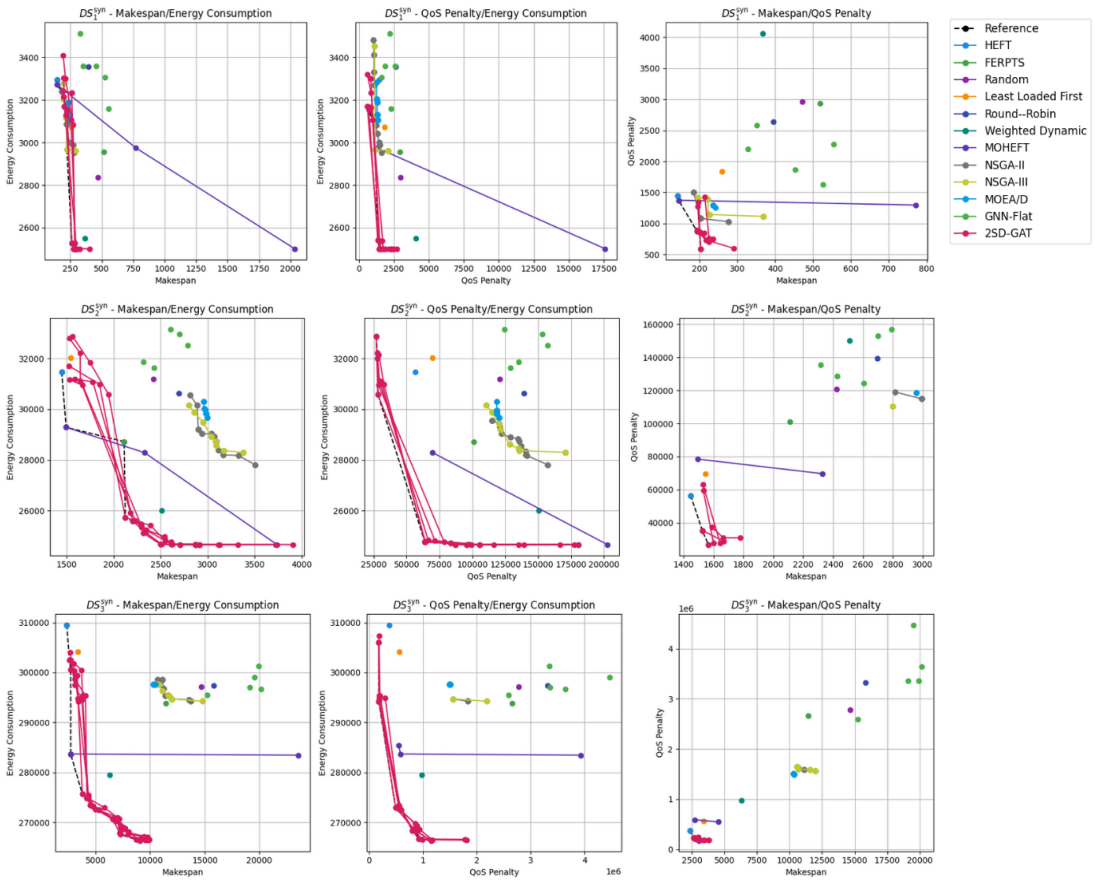

Fig. 4.5: Pareto fronts of schedulers for Synthetic datasets

Medium-scale Workload

In Figure 4.5, the Pareto front of 2SD-GAT shows a clear advantage over all baselines. Furthermore, the pareto fronts produced by 2SD-GAT consistently lie in the same region, highlighting the stochastic stability of the approach. For the Makespan/Energy projection, the front covers a broad range of trade-offs, indicating that the method can flexibly adapt to different optimization preferences. On the other two projections, 2SD-GAT solutions dominate the majority of those produced by other schedulers.

In Table 4.4 and 4.5, 2SD-GAT achieves the highest hypervolume ($(6.605 \pm 0.105) \times 10^{12}$) and the lowest IGD ($4457.338 \pm 704.960$), outperforming all other approaches. These results indicate that the Pareto front of 2SD-GAT is not only widely spread but also lies close to the reference Pareto front. The decision latency remains exceptionally low (0.083 s), significantly outperforming evolutionary-based schedulers, such as NSGA-II (22.512 s), NSGA-III (21.954 s), and MOEA/D (20.210 s). This shows that 2SD-GAT scales effectively to larger workloads while preserving its practical deployability in dynamic environments.

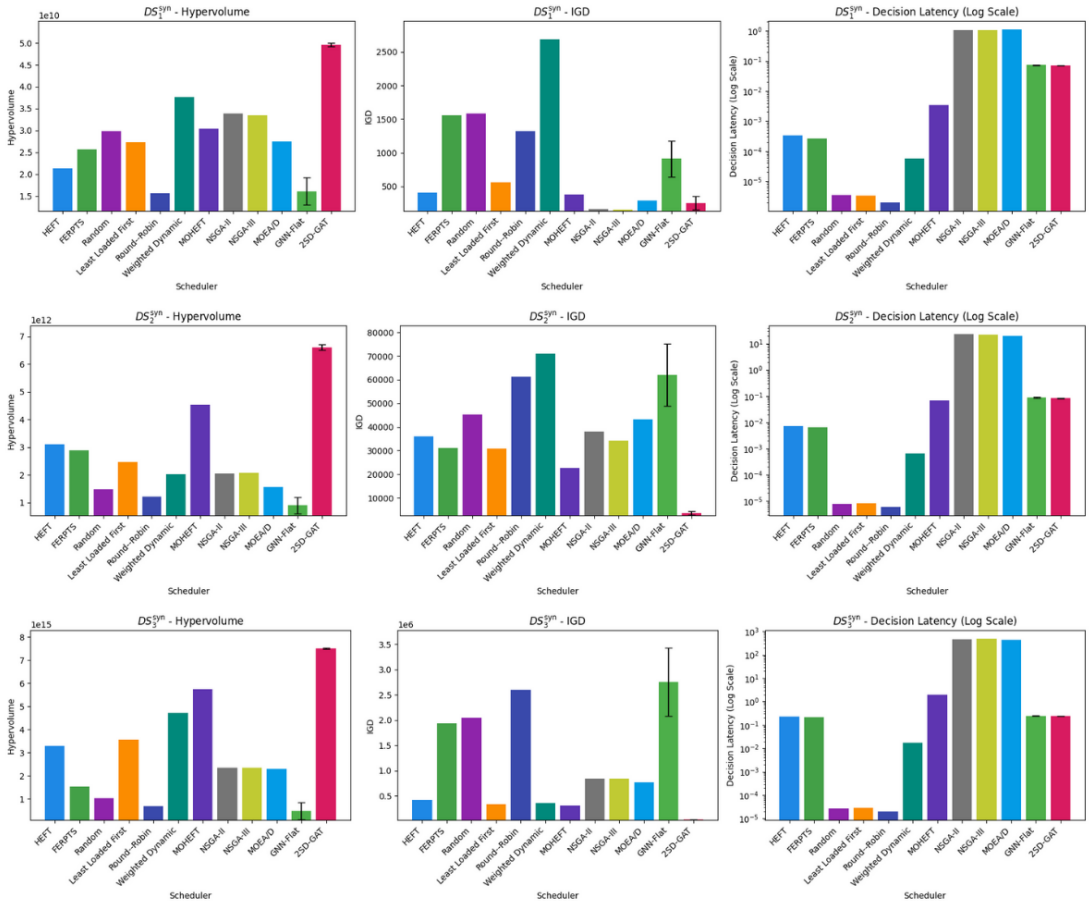

Fig. 4.6: Hypervolume/IGD/Decision Latency of schedulers for Synthetic datasets

Note. Hypervolume - higher is better, IGD/Decision Latency - lower is better

Large-scale Workload

For the largest synthetic dataset (1000 tasks), the Pareto front generated by 2SD-GAT continues to outperform the baselines, demonstrating broad coverage and strong dominance across all objective pairings. This confirms that 2SD-GAT is scalable and generalizes well beyond the conditions on which it was trained.

According to Figure 4.6, Tables 4.4 and 4.5, 2SD-GAT achieves the highest hypervolume ($(7.506 \pm 0.033) \times 10^{15}$) and the lowest IGD ($28790 \pm 3029$) among all methods, reflecting its ability to generate high-quality, diverse, and well-distributed Pareto-optimal solutions. Despite the computational complexity introduced by the large number of tasks, the decision latency remains just 0.857 s, a significant advantage over MOEA-based methods, which exhibit latencies exceeding 1000 s. At this scale, GNN-Flat exhibits substantial performance variance, likely due to incomplete convergence within the given training budget. This contrast further emphasizes the scalability of 2SD-GAT and reinforces that it is not only effective in generating optimal trade-offs but also viable for real-time use in large-scale, dynamic scheduling scenarios.

TABLE 4.4: BEST OBJECTIVE VALUES OF 2SD-GAT AND BASELINES FOR SYNTHETIC DATASETS

Dataset	Algorithm	Makespan (s)	Energy (J)	QoS Penalty
DS_1^{syn}	HEFT	**142.988**	3294.328	1448
	FERPTS	517.723	2955.773	2930
	Random	471.080	2835.402	2961
	Least Loaded First	260.105	3073.808	1836
	Round–Robin	395.300	3356.016	2639
	Weighted Dyn	366.986	<u>2549.781</u>	4064
	MOHEFT	<u>144.988</u>	**2500.936**	1298
	NSGA-II	184.335	2952.182	<u>1031</u>
	NSGA-III	195.616	2962.865	1115
	MOEA/D	236.616	3105.784	1258
	GNN-Flat	328.653	3158.429	1626
	2SD-GAT	193.335	**2500.936**	**594**
DS_2^{syn}	HEFT	**1446.927**	31463.625	<u>56536</u>
	FERPTS	2111.581	28724.176	101144
	Random	2423.312	31192.025	120784
	Least Loaded First	1544.312	32023.113	69641
	Round–Robin	2696.646	30628.496	139205
	Weighted Dyn	2510.855	<u>26020.801</u>	150227
	MOHEFT	<u>1494.927</u>	**24670.243**	69712
	NSGA-II	2814.312	27814.670	114909
	NSGA-III	2800.312	28301.329	110477
	MOEA/D	2958.312	29676.780	118479
	GNN-Flat	2317.312	31623.227	124256
	2SD-GAT	1524.773	**24670.243**	**26852**
DS_3^{syn}	HEFT	**2374.953**	309527.515	<u>378267</u>
	FERPTS	11429.292	293795.537	2657418
	Random	14679.624	297110.776	2775849
	Least Loaded First	3398.891	304178.976	561443
	Round–Robin	15832.212	297348.726	3319678
	Weighted Dyn	6312.635	<u>279493.576</u>	978795
	MOHEFT	2724.022	283468.285	548030
	NSGA-II	10683.114	294262.963	1563610
	NSGA-III	10566.920	294263.343	1563610
	MOEA/D	10316.588	297620.965	1491165
	GNN-Flat	15237.556	295475.133	2596706
	2SD-GAT	<u>2613.353</u>	**266421.366**	**178645**

Note. Best value is highlighted in **bold** text and second-best value is highlighted in <u>underline</u>.

TABLE 4.5: PARETO QUALITY METRICS OF 2SD-GAT AND BASELINES FOR SYNTHETIC DATASETS

Dataset	Algorithm	Hypervolume	IGD	Decision Latency (s)
DS_1^{syn}	HEFT	2.140×10^{10}	405.837	0.000
	FERPTS	2.566×10^{10}	1560.023	0.000
	Random	2.980×10^{10}	1586.874	0.000
	Least Loaded First	2.739×10^{10}	560.872	0.000
	Round–Robin	1.566×10^{10}	1319.864	0.000
	Weighted Dyn	3.759×10^{10}	2688.466	0.000
	MOHEFT	3.039×10^{10}	378.735	0.003
	NSGA-II	3.385×10^{10}	164.818	1.060
	NSGA-III	3.341×10^{10}	**154.817**	1.060
	MOEA/D	2.745×10^{10}	290.348	1.067
	GNN-Flat	1.609×10^{10}	912.328	0.073
	2SD-GAT	$\mathbf{4.958 \times 10^{10}}$	254.066	0.070
DS_2^{syn}	HEFT	3.098×10^{12}	44497.024	0.007
	FERPTS	2.881×10^{12}	33611.087	0.006
	Random	1.478×10^{12}	44976.566	0.000
	Least Loaded First	2.468×10^{12}	38255.143	0.000
	Round–Robin	1.207×10^{11}	57236.619	0.000
	Weighted Dyn	2.016×10^{12}	65281.783	0.000
	MOHEFT	4.526×10^{12}	22450.594	0.001
	NSGA-II	2.052×10^{12}	35135.431	22.512
	NSGA-III	2.065×10^{12}	31445.528	21.954
	MOEA/D	1.565×10^{12}	43063.644	20.210
	GNN-Flat	8.922×10^{11}	54660.148	0.087
	2SD-GAT	$\mathbf{6.605 \times 10^{12}}$	**4457.338**	0.083
DS_3^{syn}	HEFT	3.308×10^{15}	424177	0.228
	FERPTS	1.529×10^{15}	1933543	0.220
	Random	1.044×10^{14}	2052007	0.000
	Least Loaded First	3.561×10^{15}	337402	0.000
	Round–Robin	6.951×10^{14}	2595804	0.000
	Weighted Dyn	4.709×10^{15}	359445	0.017
	MOHEFT	5.748×10^{15}	314396	1.950
	NSGA-II	2.343×10^{15}	840378	473.643
	NSGA-III	2.358×10^{15}	840378	478.179
	MOEA/D	2.290×10^{15}	770403	435.758
	GNN-Flat	4.771×10^{14}	2760824	0.247
	2SD-GAT	$\mathbf{7.506 \times 10^{15}}$	**28790**	0.244

Note. Best value is highlighted in **bold** text and second-best value is highlighted in underline.

4.2.1.1 Results for Real and Dynamic Datasets

This section presents an evaluation of the proposed scheduling algorithm against a range of baseline methods under real-world workload conditions, both with and without

dynamic disruptions. Table 4.6 and Table 4.7 reports the quantitative results for these two datasets. Corresponding visualizations are provided in Figures 4.7 and 4.8, which illustrate the Pareto fronts and multi-objective metric comparisons. Two versions of the proposed agent are evaluated: the agent trained on synthetic data, denoted as 2SD-GAT (Syn), and the agent trained on real-world data without dynamic scenarios, denoted as 2SD-GAT (Real).

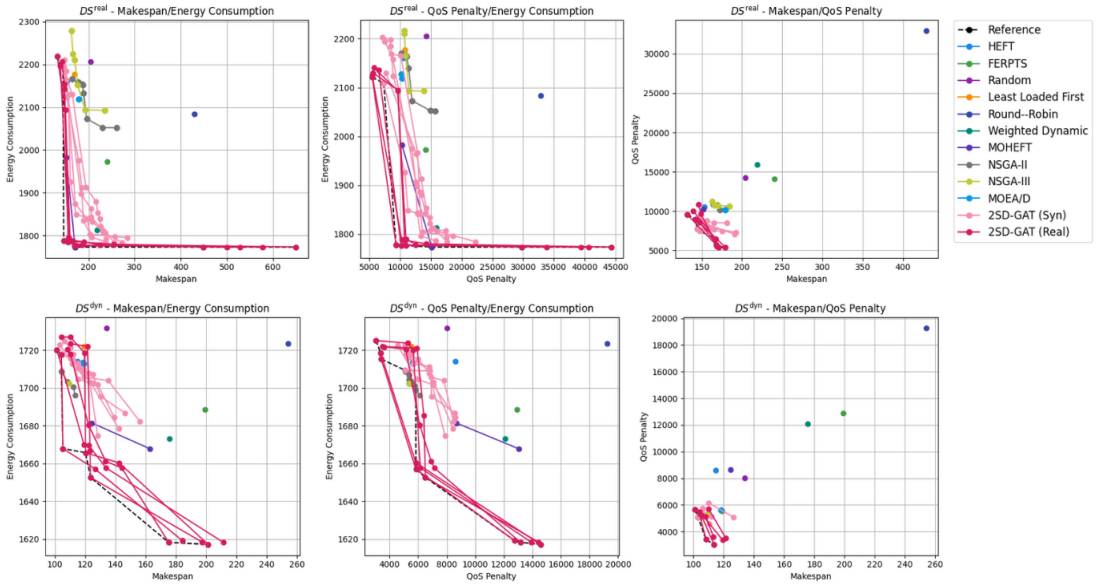

Fig. 4.7: Pareto fronts of schedulers for Real and Dynamic datasets

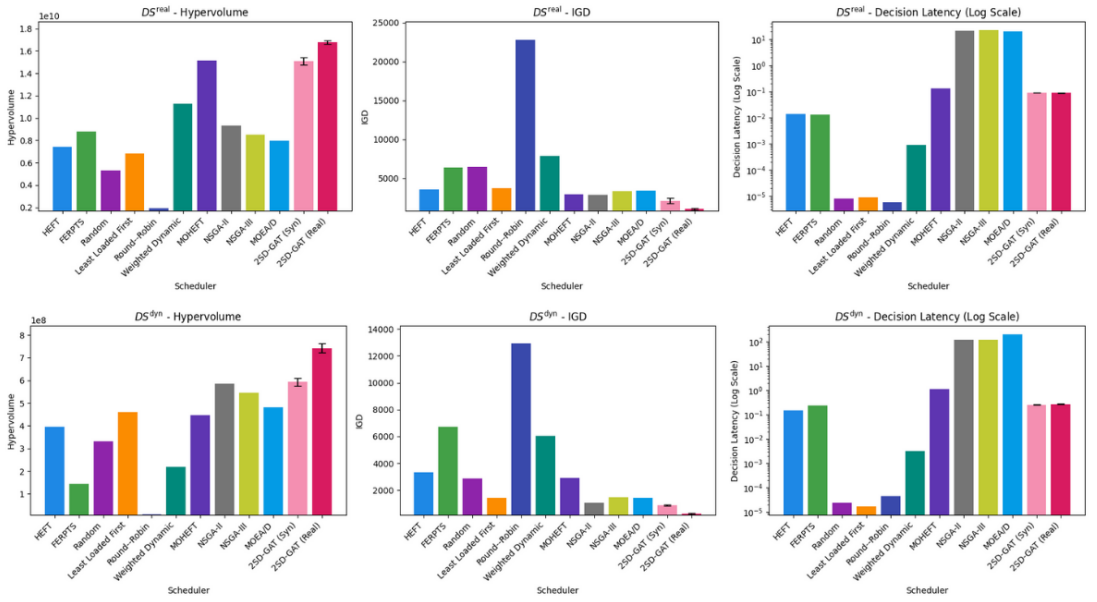

Fig. 4.8: Hypervolume of Schedulers for Real and Dynamic datasets

Note. Hypervolume - higher is better, IGD/Decision Latency - lower is better

TABLE 4.6: BEST OBJECTIVE VALUES OF 2SD-GAT AND BASELINES FOR REAL-WORLD DATASETS

Dataset	Algorithm	Makespan (s)	Energy (J)	QoS Penalty
DS^{real}	HEFT	153.561	2160.173	10549
	FERPTS	240.197	1972.795	14128
	Random	204.197	2205.772	14236
	Least Loaded First	169.248	2177.238	10798
	Round–Robin	429.022	2084.045	32894
	Weighted Dyn	218.939	1812.386	15916
	MOHEFT	151.756	**1773.266**	10278
	NSGA-II	164.197	2052.155	10184
	NSGA-III	163.022	2092.714	10660
	MOEA/D	179.197	2119.260	10194
	2SD-GAT (Syn)	<u>144.545</u>	<u>1783.226</u>	<u>7059</u>
	2SD-GAT (Real)	**131.939**	**1773.266**	**5442**
DS^{dyn}	HEFT	114.939	1713.977	8589
	FERPTS	199.229	1688.687	12910
	Random	133.965	1731.580	8029
	Least Loaded First	119.229	1721.968	5510
	Round–Robin	254.229	1723.616	19250
	Weighted Dyn	175.626	<u>1673.257</u>	12077
	MOHEFT	124.471	1667.807	8670
	NSGA-II	104.229	1696.270	5096
	NSGA-III	109.229	1702.322	5351
	MOEA/D	118.229	1712.737	5623
	2SD-GAT (Syn)	<u>103.229</u>	1674.673	<u>4572</u>
	2SD-GAT (Real)	**101.229**	**1617.236**	**3045**

Note. Best value is highlighted in **bold** text and second-best value is highlighted in <u>underline</u>.

Real-World Dataset

In Figure 4.7, the Pareto front generated by the 2SD-GAT (Real) model dominates the majority of solutions across all baseline methods, exhibiting strong spread and high-quality trade-offs across objective pairs. The 2SD-GAT (Syn) model, trained only on synthetic data, also performs competitively and dominates several classical and heuristic approaches. However, it shows a marginally lower performance compared to the 2SD-GAT (Real) model. This result is expected and likely stems from distribution shift. The synthetic training data differs in structure and workflow patterns, as it relies on Erdős-Rényi $G(n, p)$ graphs, while the real dataset consists of branch–parallel–merge DAGs that better reflect practical cloud workloads. Such differences naturally degrade IGD when a model is evaluated under domain shift rather than indicating strict dataset specificity.

TABLE 4.7: PARETO QUALITY METRICS OF 2SD-GAT AND BASELINES FOR REAL-WORLD DATASETS

Dataset	Algorithm	Hypervolume	IGD	Decision Latency (s)
DS^{real}	HEFT	7.412×10^9	3598.126	0.013
	FERPTS	8.761×10^9	6356.601	0.013
	Random	5.289×10^9	6449.943	0.000
	Least Loaded First	6.805×10^9	3749.081	0.000
	Round–Robin	1.912×10^9	22836.320	0.000
	Weighted Dyn	1.129×10^{10}	7851.135	0.001
	MOHEFT	$\underline{1.517 \times 10^{10}}$	2962.564	0.131
	NSGA-II	9.352×10^9	2870.509	21.223
	NSGA-III	8.539×10^9	3353.743	21.780
	MOEA/D	7.984×10^9	3410.190	20.093
	2SD-GAT (Syn)	1.509×10^{10}	<u>2108.377</u>	0.090
	2SD-GAT (Real)	$\mathbf{1.678 \times 10^{10}}$	**1057.388**	0.089
DS^{dyn}	HEFT	3.955×10^8	3320.501	0.148
	FERPTS	1.436×10^8	6727.567	0.237
	Random	3.316×10^8	2880.840	0.000
	Least Loaded First	4.593×10^8	1433.281	0.000
	Round–Robin	8.864×10^6	12930.079	0.000
	Weighted Dyn	2.191×10^8	6061.232	0.003
	MOHEFT	4.462×10^8	2935.953	1.131
	NSGA-II	5.855×10^8	1066.093	116.829
	NSGA-III	5.459×10^8	1478.809	117.356
	MOEA/D	4.820×10^8	1408.739	194.476
	2SD-GAT (Syn)	$\underline{5.933 \times 10^8}$	<u>896.610</u>	0.254
	2SD-GAT (Real)	$\mathbf{7.425 \times 10^8}$	**237.332**	0.272

Note. Best value is highlighted in **bold** text and second-best value is highlighted in <u>underline</u>.

These results validate the benefit of fine-tuning the RL agent on problem-specific data. The 2SD-GAT (Real) model, although trained on a small-scale real dataset (with 50 tasks), generalizes effectively to the 250-task workload seen in this evaluation, indicating that the architecture can adapt well to the target domain.

From Figure 4.8 and Tables 4.6 and 4.7, the 2SD-GAT (Real) method achieves the best hypervolume ($(1.678 \pm 0.014) \times 10^{10}$) and IGD ($2108.377 \pm 347.929$), indicating superior Pareto front quality. The 2SD-GAT (Syn) model follows closely, with the hypervolume ($(1.509 \pm 0.032) \times 10^{10}$) and second-best IGD ($1057.388 \pm 115.391$). Importantly, despite being trained on a different distribution, the synthetic-trained model remains the second-best overall, suggesting reasonable cross-domain generalization. Both models maintain decision latencies well below 0.1 s, significantly outperforming evolutionary baselines like NSGA-II (21.223 s) and MOEA/D (20.093 s), reinforcing

the practicality of the 2SD-GAT approach for real-time scheduling.

Dynamic Real-World Dataset

The dynamic variant of the real dataset introduces runtime uncertainties through simulated VM failures and inaccurate task length estimations. As shown in Figure 4.7, the 2SD-GAT (Real) model continues to maintain strong performance, with a Pareto front that dominates most of the baselines and demonstrates a wide spread across all projections. The 2SD-GAT (Syn) model also performs well, reinforcing its robustness despite not being explicitly trained on dynamic scenarios.

In contrast, static scheduling algorithms struggle to adapt in the face of environmental changes, requiring frequent re-computation and rescheduling. This is reflected in their performance metrics: MOHEFT, NSGA-II, and MOEA/D exhibit long decision latencies ranging from 1 s to over 190 s. Their IGD scores also degrade, indicating poorer solution quality under dynamic conditions.

As reported in Table 4.7, the 2SD-GAT (Real) model achieves the best hypervolume ($(7.425 \pm 0.207) \times 10^8$) and IGD ($237.332 \pm 48.518$), demonstrating strong adaptability and robustness to environmental changes. The 2SD-GAT (Syn) model attains the second-best hypervolume ($(5.933 \pm 0.170) \times 10^8$) and a comparable IGD ($237.332 \pm 48.518$), indicating that even when trained on datasets with different statistical distributions and workflow structures, the model generalizes effectively. Although the fine-tuned 2SD-GAT (Real) outperforms 2SD-GAT (Syn), the performance gap remains modest. NSGA-II achieves the third-best hypervolume (5.855×10^8), suggesting that it generates a well-distributed set of solutions across the objective space. However, its substantially higher IGD (1066.093) indicates that these solutions lie farther from the true Pareto front, reflecting weaker convergence despite good diversity. Furthermore, its extreme decision latency (exceeding 3 minutes) limits its practicality for real-time or dynamic scheduling scenarios. In contrast, both 2SD-GAT (Syn) and 2SD-GAT (Real) outperform all other baselines in overall solution quality and decision latency, making them more suitable for practical deployment.

4.2.2 Ablation Study

To assess the contribution of the GAT-based encoder within the proposed 2SD-GAT framework, an ablation study was conducted by replacing the GAT component of the Task Actor with two alternative architectures: a GIN [26] and a MLP. These variants, denoted as 2SD-GIN and 2SD-MLP respectively, preserve all other components of the two-stage scheduling architecture. This ensures that any performance differences arise solely from the choice of graph encoder used to process workflow dependencies.

The GIN baseline represents a powerful message-passing GNN with injective aggregation, enabling it to distinguish structurally different neighborhoods. However,

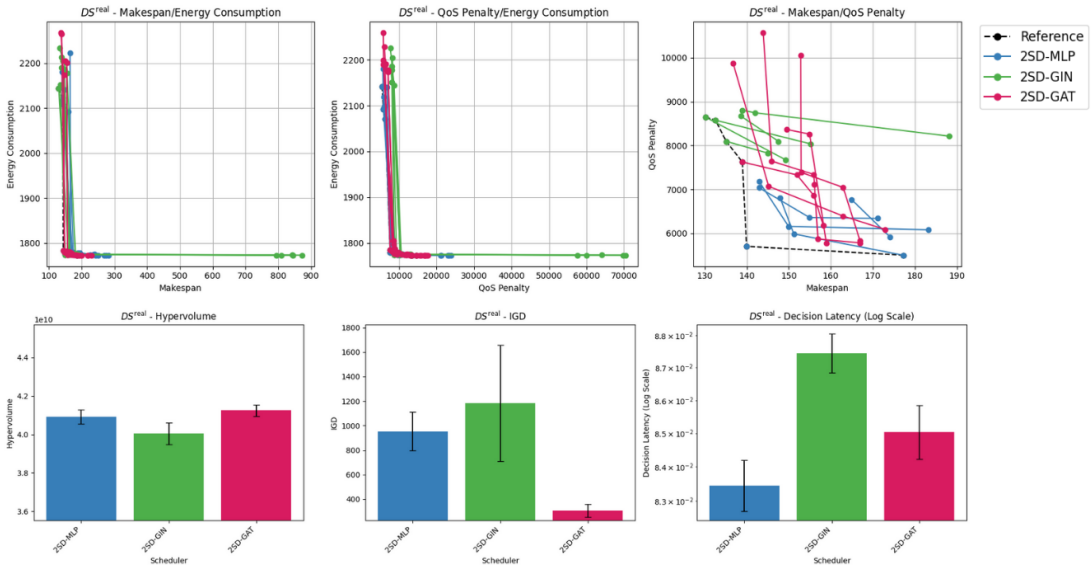

Fig. 4.9: Pareto fronts/Hypervolume/IGD/Decision Latency of ablation study

Note. Hypervolume - higher is better, IGD/Decision Latency - lower is better

unlike GAT, it applies uniform weights to all neighbors during aggregation and therefore cannot modulate the relative importance of different dependencies. The MLP baseline removes explicit graph modeling entirely, operating only on per-task feature vectors and relying on the DRL agent to learn implicit structural cues without a dedicated message-passing mechanism. Both alternatives constitute meaningful points of comparison for isolating the benefits of attention-based aggregation.

All three architectures, 2SD-GAT, 2SD-GIN, and 2SD-MLP, were evaluated using the DS^{real} real-world dataset configuration described in Section 3.3.8, ensuring that each model encounters identical workflow structures, task characteristics, and VM configurations during evaluation. The detailed results of this ablation study are reported in Table 4.8, with Figure 4.9 providing a visual comparison.

The quantitative results reveal several key insights. First, while 2SD-MLP achieves competitive performance in certain single-objective metrics such as best QoS penalty (5502), it consistently falls behind 2SD-GAT in MO performance. Specifically, 2SD-GAT achieves the highest hypervolume ($(4.125 \pm 0.031) \times 10^{10}$) and the lowest IGD ($306.060 \pm 51.785$), indicating substantially better Pareto-front coverage and convergence. In contrast, 2SD-MLP, despite being the second-best performer, attains a noticeably weaker IGD (954.055 ± 157.106) and a lower hypervolume ($(4.092 \pm 0.036) \times 10^{10}$).

The 2SD-GIN variant performs noticeably worse than both 2SD-GAT and 2SD-MLP across all MO metrics, with the lowest hypervolume ($(4.005 \pm 0.055) \times 10^{10}$), the highest IGD ($1184 \pm 473.192$), and the worst QoS penalty (7675). This degradation is consistent with the architectural limitations of GIN: while it preserves expressive neighborhood aggregation, its uniform weighting prevents it from differentiating between more and

TABLE 4.8: Performance Comparison of 2SD-GAT, 2SD-GIN, and 2SD-MLP schedulers (Best value is highlighted in **bold** text and second-best value is highlighted in underline)

Dataset	Algorithm	Best Makespan (s)	Best Enegy Consumption (J)	Best QoS Penalty
DS^{real}	2SD-MLP	139.939	**1773.266**	**5502**
	2SD-GIN	**130.197**	**1773.266**	7675
	2SD-GAT	136.721	**1773.266**	5777
Dataset	**Algorithm**	**Hypervolume**	**IGD**	**Decision Latency (s)**
DS^{real}	2SD-MLP	4.092×10^{10}	954.055	**0.083**
	2SD-GIN	4.005×10^{10}	1184.586	0.087
	2SD-GAT	$\mathbf{4.125 \times 10^{10}}$	**306.060**	0.085

less critical dependencies, an ability that is particularly important in workflow DAGs with heterogeneous structural bottlenecks.

All approaches exhibit nearly identical average decision latencies (0.083, 0.087, and 0.085), indicating that the added expressiveness of attention-based aggregation does not introduce measurable runtime overhead. In practice, this shows that 2SD-GAT maintains competitive inference efficiency while delivering substantially higher solution quality.

Overall, the ablation findings reinforce the suitability of GAT as the encoder within the two-stage DRL architecture.

4.2.3 Sensitivity Analysis

TABLE 4.9: Linear sensitivity models relating preference weights $(\lambda_1, \lambda_2, \lambda_3)$ to performance metrics.

Metric	β_0	β_M	β_E	β_Q	R^2
Makespan	98.6015	-6.8887	17.3124	-5.7111	0.5955
Energy Consumption	462.0045	1.9710	-40.6047	1.4411	0.7522
QoS Penalty	1189.9709	24.1977	461.7810	-263.6079	0.6358
Decision Latency	0.0923	-0.0001	0.0015	0.0019	0.0656

In preference-based multi-objective reinforcement learning, it is important to verify that the learned policies respond consistently to changes in the preference weights used during training. Since each agent in the proposed framework is trained using a fixed preference vector $(\lambda_1, \lambda_2, \lambda_3)$, the expected behaviour is that increasing the weight assigned to a particular objective should steer the learned policy toward performance outcomes that reflect that objective's preference. To assess whether this holds in practice, a post-hoc sensitivity analysis was conducted on the trained agents.

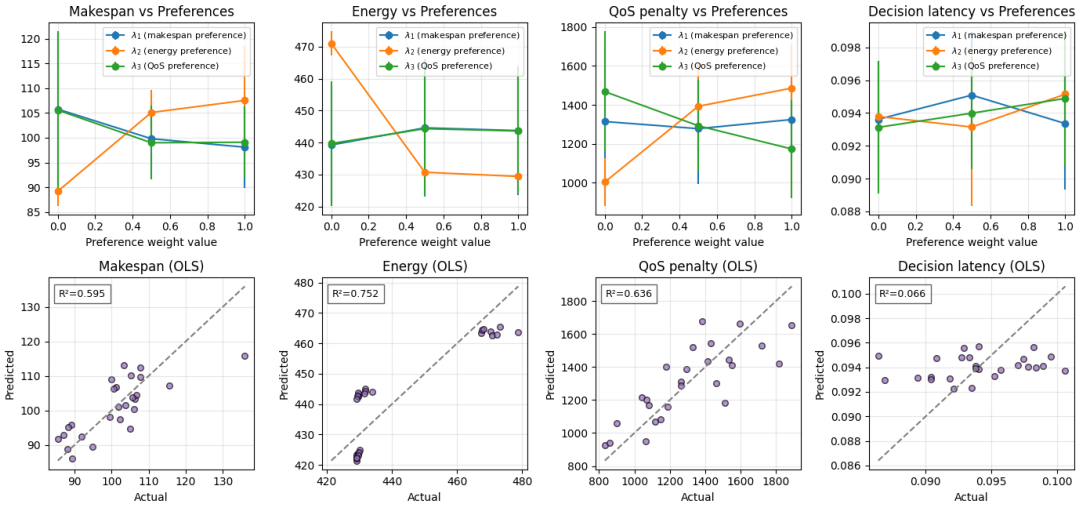

Fig. 4.10: Sensitivity of agent performance to preference weights (Top) and Sensitivity analysis of agent behaviour under varying preference weights (bottom).

Note. Hypervolume - higher is better, IGD/Decision Latency - lower is better

The analysis considers the mapping from preference weights to the resulting performance metrics: makespan, energy consumption, and QoS penalty. A total of 26 agents trained under different preference tuples were evaluated on the same test workflows, producing paired data of the form $(\lambda_1, \lambda_2, \lambda_3) \mapsto (M, E, Q)$. To quantify how strongly each objective responds to its corresponding preference weight, we fit a separate linear regression model for each performance metric, using the preference weights as independent variables. This yields a first-order approximation as given in eq. (4.1):

$$y = \beta_0 + \beta_M \lambda_1 + \beta_E \lambda_2 + \beta_Q \lambda_3, \tag{4.1}$$

The resulting regression coefficients are reported in Table 4.9, with Figure 4.10 providing a visual interpretation of the linear trends.

Quantitatively, the makespan model shows that λ_1 exerts a negative influence ($\beta_M = -6.89$), confirming that stronger makespan preference consistently drives the policy toward faster schedule completion. In contrast, the positive coefficient for λ_2 ($\beta_E = 17.31$) reveals that energy-aware agents tend to produce longer schedules, a natural consequence of adopting more conservative or energy-efficient allocation strategies.

Energy consumption exhibits the strongest and most interpretable sensitivity patterns. The large negative coefficient associated with λ_2 ($\beta_E = -40.60$) shows that agents prioritising energy substantially reduce resource usage. Meanwhile, the smaller positive coefficients for λ_1 and λ_3 (1.97 and 1.44, respectively) indicate that agents focusing on makespan or QoS tend to use slightly more resources. The high coefficient of

determination ($R^2 = 0.75$) suggests that the preference weights explain most of the variation in energy usage.

QoS penalty responds strongly to both λ_2 and λ_3. The large negative coefficient for λ_3 ($\beta_Q = -263.61$) shows that increasing the QoS preference dramatically reduces QoS penalties, while the positive coefficient for λ_2 ($\beta_E = 461.78$) indicates that energy-efficient scheduling significantly increases QoS penalties. This trade-off aligns with the expected behaviour of agents operating under resource-saving constraints.

Decision latency exhibits negligible sensitivity to the preference weights, as reflected by the extremely small coefficients (all on the order of 10^{-3} or smaller) and the low R^2 value (0.0656). This indicates that the choice of preferences does not materially affect the time required for the agent to compute scheduling decisions, suggesting that latency is dominated by architectural factors rather than behavioural ones.

Overall, the sensitivity analysis demonstrates that the 2SD-GAT framework responds in a stable and interpretable manner to changes in the preference vector.

4.2.4 Computational Cost and Training Time

All experiments were conducted on the hardware described in Section 3.3.10. Each agent was trained for 200,000 environment steps. Across all trained preference-conditioned agents, the average training time per agent for the DS^{real} was 16.07 ± 1.18 minutes. Since the Pareto front is constructed using 26 agents with different preference vectors, the total sequential training budget is about 7 hours. This cost scales linearly with the number of preference vectors but can be reduced substantially through parallel training, as agents are fully independent. Importantly, this training cost is incurred offline only once. At inference time, the learned policies produce scheduling decisions with an average latency of ≈ 0.067 s per decision for the DS^{real}, which is negligible compared to typical workflow execution times. Therefore, the proposed approach maintains practical deployability while enabling broad Pareto-front coverage.

4.3 Results for Scheduler Implementations and Integrations

4.3.1 Results for Apache Airflow Integration

This section presents the experimental results obtained from integrating the proposed RL-driven scheduling framework with Apache Airflow. The performance of the learned scheduler (2SD-GAT) is compared against Airflow's default scheduling mechanism across eleven workflow instances. Evaluation focuses on three primary performance metrics: makespan, energy consumption, and QoS penalty. Table 4.10 summarizes the quantitative results, while Figure 4.11 provides a visual comparison across instances.

Across all experiment instances, the RL-driven scheduler consistently outperformed the default Airflow scheduler in terms of execution efficiency. The most notable

TABLE 4.10: COMPARISON OF RL AND DEFAULT ACROSS EXPERIMENT INSTANCES

Instance	Algorithm	Makespan (s)	Energy Consumption (J)	QoS Penalty
0	Airflow Default	120	363.152	1501
	RL (2SD-GAT)	**101**	**208.316**	**949**
1	Airflow Default	128	**338.744**	1650
	RL (2SD-GAT)	**84**	**338.744**	**1010**
2	Airflow Default	74	**572.476**	781
	RL (2SD-GAT)	**61**	**572.476**	**431**
3	Airflow Default	222	**337.445**	2897
	RL (2SD-GAT)	**159**	**337.445**	**1872**
4	Airflow Default	117	281.672	1320
	RL (2SD-GAT)	**68**	**277.658**	**1061**
5	Airflow Default	100	363.152	1501
	RL (2SD-GAT)	**80**	**208.316**	**949**
6	Airflow Default	103	**338.744**	1650
	RL (2SD-GAT)	**71**	**338.744**	**1010**
7	Airflow Default	78	**572.476**	781
	RL (2SD-GAT)	**51**	**572.476**	**431**
8	Airflow Default	97	281.672	1320
	RL (2SD-GAT)	**67**	**277.658**	**1061**
9	Airflow Default	136	314.654	2228
	RL (2SD-GAT)	**98**	**245.967**	**1575**

Note. Best value (lower is better) is highlighted in **bold**.

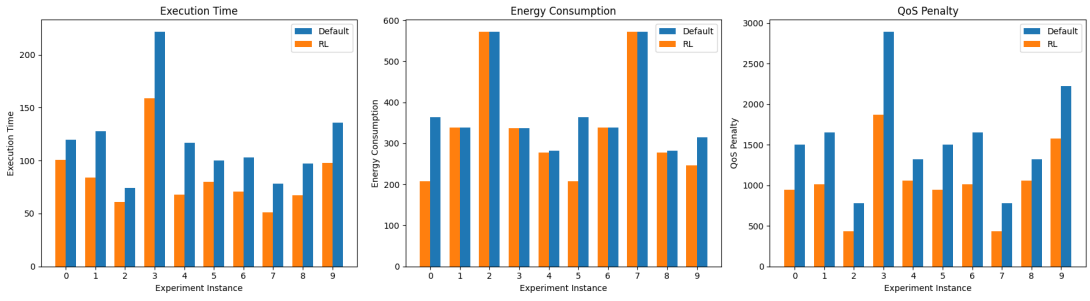

Fig. 4.11: Comparison of RL and Default Schedulers Across Experiment Instances for Airflow Integration

improvements were observed in makespan and QoS penalty, indicating that the learned policy effectively optimized task ordering and worker allocation despite operating within Airflow's static DAG constraints.

The RL scheduler achieved lower makespan values in all evaluated instances, demonstrating stable performance improvements independent of workflow variation. Large

workflows benefited particularly strongly from adaptive planning. For example, instances with higher baseline makespans (e.g., instances 3 and 9) experienced substantial reductions. Even in smaller workflows with relatively short baseline runtimes, the RL scheduler maintained measurable performance gains.

Energy consumption exhibited more nuanced behaviour. While several instances showed clear reductions, others produced identical energy usage between the two approaches. Since the number of hosts in the dataset is small (only three), energy consumption is primarily determined by the total number of tasks and their execution durations, which remain similar across both schedulers. This suggests that although the RL scheduler improves task ordering and resource allocation, it does not necessarily reduce energy consumption in comparable workflows with limited resource configurations. Importantly, the absence of energy regressions demonstrates that integrating adaptive planning does not introduce additional computational overhead at runtime.

Similarly, QoS penalties were reduced for every instance, indicating improved deadline adherence and reduced scheduling inefficiencies. The magnitude of improvement was particularly pronounced in complex workflows, where inefficient default scheduling produced large penalties. By enforcing execution order through RL-generated dependency edges, the system minimized scheduler-induced delays while preserving Airflow's deterministic execution guarantees.

Overall, these results confirm that introducing a pre-execution planning layer can effectively overcome limitations imposed by Airflow's static scheduling model without requiring modifications to its internal scheduler.

4.3.2 Results for Kubernetes-Based Scheduling Integration

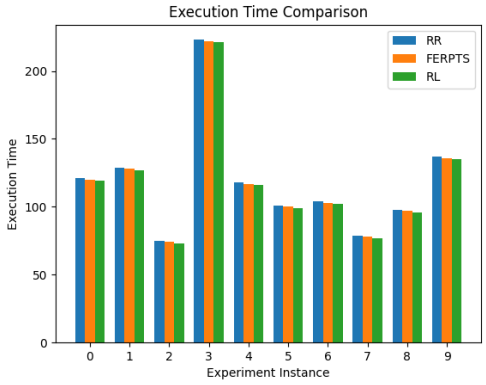

Fig. 4.12: Comparison of RL and Default Schedulers Across Experiment Instances for Kubernetes Integration

This section presents the performance evaluation of the proposed RL-based scheduling framework integrated directly with Kubernetes. The results compare three schedul-

ing approaches: Round Robin, FERPTS, and the proposed RL-based scheduler (2SD-GAT). Figure 4.12 illustrates the execution time observed across the experiment instances.

Across all evaluated experiments, a consistent ordering of performance was observed. The Round Robin scheduler produced the highest execution times, followed by FERPTS, while the 2SD-GAT scheduler achieved the lowest execution times in each case. Although the differences between algorithms were not large, the results demonstrate a stable improvement trend in favour of the proposed method.

The relatively small performance gap can be attributed primarily to the operational characteristics of Kubernetes. Unlike workflow engines such as Airflow, Kubernetes introduces additional runtime overhead through pod creation, container initialization, and scheduling latency. These factors contribute significantly to total execution time and reduce the observable impact of improved task placement decisions.

4.3.3 Results for the RL-Based Pluggable Scheduler Prototype

This section presents the experimental results obtained using the RL-based pluggable scheduler prototype. The objective of the evaluation is to compare the performance of the proposed DRL scheduling strategy against baseline scheduling algorithms under identical execution conditions. The evaluated algorithms include Round Robin, Random scheduling, HEFT, and the proposed DRL-based scheduler. Performance is assessed using three primary metrics: workflow makespan, total energy consumption, and QoS penalty.

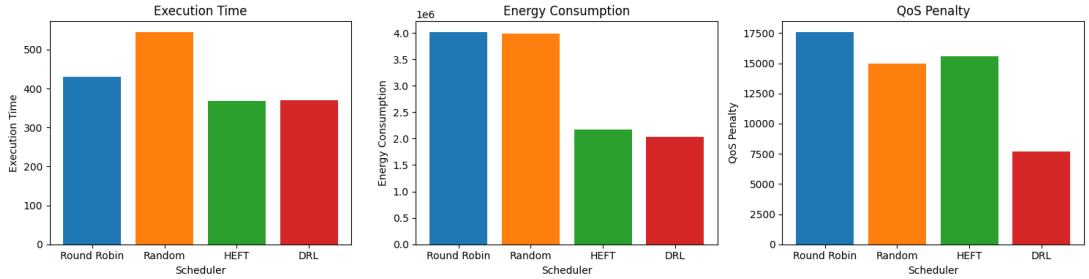

Fig. 4.13: Comparison of task execution timelines for the proposed scheduler prototype under different scheduling algorithms

Table 4.11 and Figure 4.13 summarize the aggregated results across all experiments. Lower values indicate better performance for all reported metrics.

The HEFT scheduler achieves the lowest makespan of 367.8012 s, closely followed by the DRL scheduler with a makespan of 370.3899 s. Both approaches significantly outperform the Round Robin and Random schedulers, which produce substantially longer execution times. The Random scheduler exhibits the highest makespan, indicating inefficient task placement decisions under dynamic workflow conditions.

TABLE 4.11: COMPARISON OF SCHEDULING ALGORITHMS

Algorithm	**Makespan (s)**	**Energy Consumption (J)**	**QoS Penalty**
Round Robin	429.4270	4017447.7750	17590.2553
Random	545.2233	3985176.5566	<u>14949.0217</u>
HEFT	**367.8012**	<u>2167853.3200</u>	15548.3310
DRL	<u>370.3899</u>	**2033103.9306**	**7705.2847**

Note. Best value (lower is better) is highlighted in **bold**, and second-best value is highlighted in <u>underline</u>.

In terms of energy consumption, the DRL scheduler achieves the best performance with a total energy usage of 2,033,103.9306 J, representing a considerable reduction compared to all baseline methods. HEFT provides the second-best energy efficiency, while Round Robin and Random scheduling consume nearly twice as much energy, reflecting poorer alignment between task assignments and worker capabilities.

The most pronounced difference is observed in QoS penalty values. The DRL scheduler produces the lowest QoS penalty (7705.2847), significantly outperforming all competing algorithms. Random scheduling achieves the second-best QoS penalty despite its poor makespan performance, whereas Round Robin results in the highest accumulated penalty.

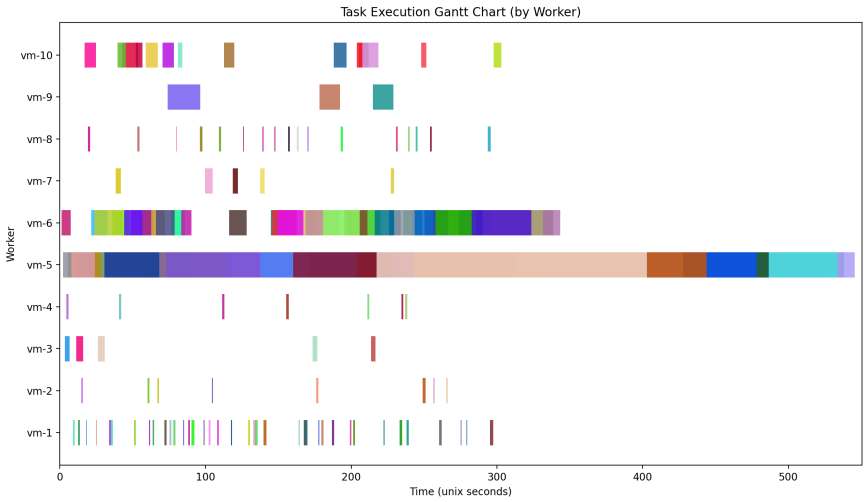

Fig. 4.14: Gantt chart of task execution timelines for the proposed scheduler prototype under Random scheduler

The Random scheduler (Figure 4.14) shows fragmented execution with uneven worker load distribution and noticeable idle intervals. Task assignments appear uncorrelated with execution duration or dependencies, leading to inefficient workflow progression.

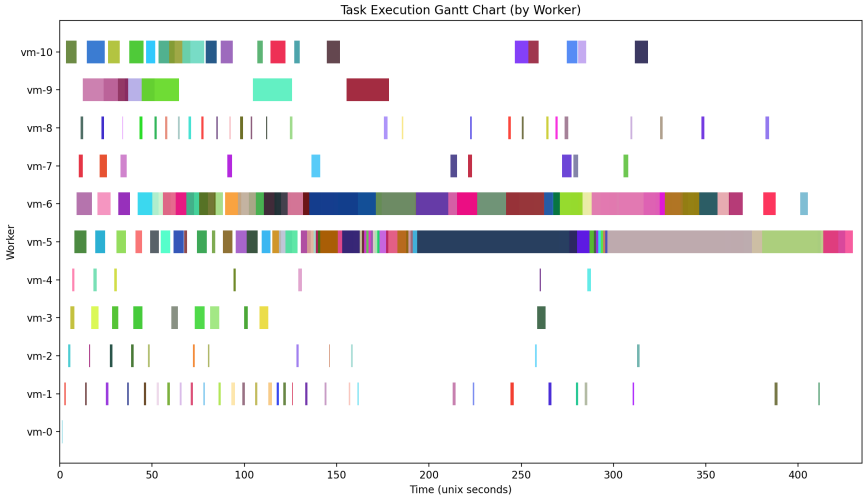

Fig. 4.15: Gantt chart of task execution timelines for the proposed scheduler prototype under Round-Robin scheduler

The Round Robin scheduler (Figure 4.15) demonstrates balanced task distribution but lacks awareness of task heterogeneity. Long-running tasks occasionally block faster workers, increasing overall completion time.

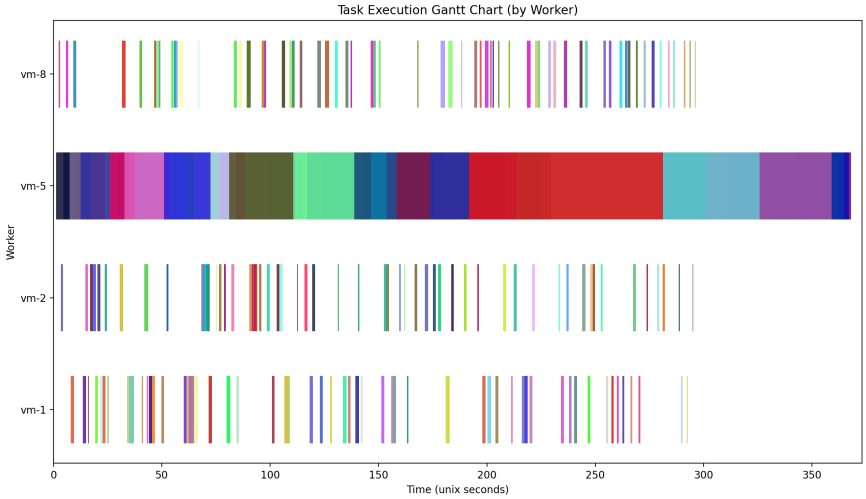

Fig. 4.16: Gantt chart of task execution timelines for the proposed scheduler prototype under HEFT scheduler

HEFT scheduling (Figure 4.16) results in more ordered execution, where dependency-aware prioritization reduces idle time and accelerates critical task completion. Worker utilization becomes more consistent, contributing to improved makespan performance.

The DRL-based scheduler (Figure 4.17) produces the most compact pattern. Tasks are distributed in a manner that maintains sustained worker activity while minimizing prolonged idle periods. Concurrent execution is preserved without excessive contention, indicating adaptive scheduling decisions based on observed system state. Here, we can

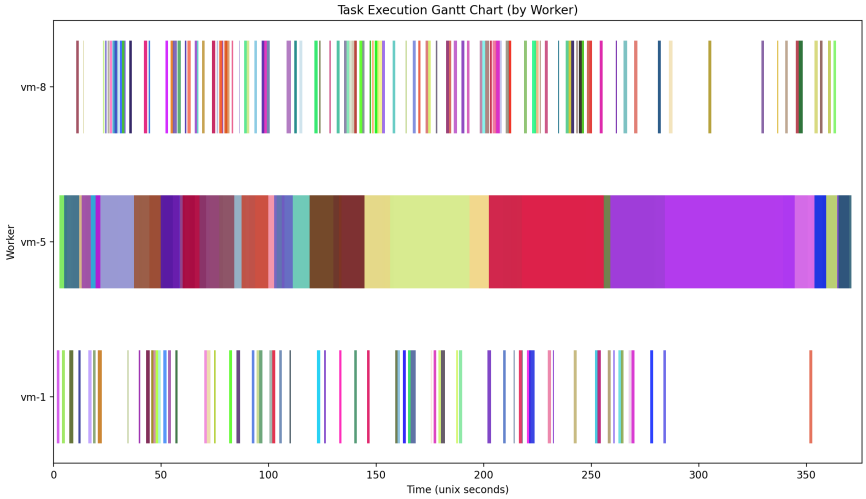

Fig. 4.17: Gantt chart of task execution timelines for the proposed scheduler prototype under Weighted Dynamic scheduler

see that the scheduler has preferred specific VMs that have lower energy consumption, which is likely a key factor contributing to the observed energy efficiency improvements.

Across all evaluated metrics, heuristic and learning-based schedulers outperform non-adaptive baselines. HEFT achieves the best execution time, while the DRL scheduler provides the best energy efficiency and QoS performance, achieving near-optimal makespan simultaneously. These results demonstrate that the pluggable prototype successfully enables comparative evaluation of scheduling strategies and that the RL-based approach operates competitively with established heuristic methods within the experimental environment.

CHAPTER 5

DISCUSSION

This chapter provides a comprehensive discussion of the methodology, contributions, evaluation, and applicability of the proposed research on energy-efficient cloud workflow scheduling using DRL.

As a summary, this research proposes a multi-objective DRL-based framework for cloud workflow scheduling that optimizes three critical objectives: makespan, energy consumption, and QoS latency (addresses **RQ1**). Workflows are modeled as DAGs, and a graph-based agent trained using the PPO algorithm is developed to make scheduling decisions. Two agent variants were explored: a flat GNN-based agent using GIN, and an extended two-stage model using GAT (covers **RQ2**). Beyond algorithmic performance, the work also investigates how such learning-based schedulers can be realistically integrated into existing workflow orchestration and cloud-native execution environments (addresses **RQ4**).

5.1 Baseline Research

The design of the proposed model is inspired by recent advances in applying GNNs and DRL to structured scheduling problems such as the FJSSP. In particular, Lei et al. [56] demonstrated that using GNNs with multi-action DRL significantly improves scheduling performance in complex job-shop environments with dynamic constraints and multiple compatible machines. Their architecture employs a disjunctive graph representation and a dual-policy agent trained using a multi-action version of PPO, achieving strong generalization even in large-scale settings.

The architecture consists of two coordinated agents: one selects the operation to schedule, and the other assigns it to a compatible machine. These agents operate on embeddings generated via a disjunctive graph, capturing precedence constraints and machine eligibility. The framework significantly outperforms both heuristic and metaheuristic baselines across random and benchmark instances, demonstrating strong generalization on large-scale scenarios.

The Lei et al. [56] model is selected as the baseline due to its structural relevance to cloud workflow scheduling. FJSSP and cloud workflow scheduling share foundational similarities, as both involve allocating interdependent tasks or operations to a limited set of compatible resources while adhering to precedence constraints and capacity limitations. In both domains, tasks must be mapped to resources that satisfy their execution requirements, the environments are inherently heterogeneous and constrained, and the optimization objectives revolve around improving performance metrics such as makespan and ensuring efficient resource usage including energy efficiency.

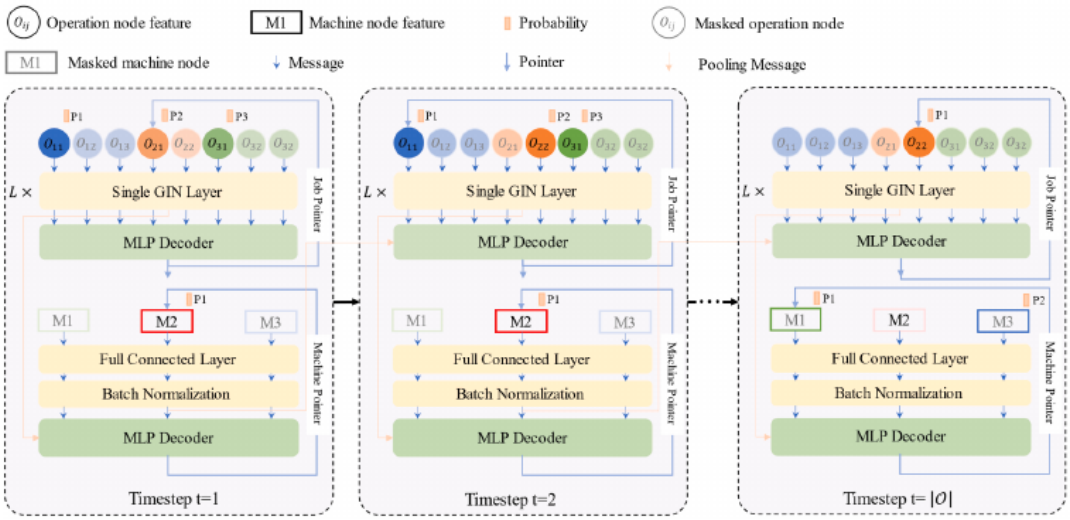

Fig. 5.1: The MPGN architecture for the FJSSP

Note. Reproduced from [56]

The success of Lei et al.'s GNN-enhanced DRL architecture in solving large-scale, dynamic scheduling tasks makes it a strong candidate for adaptation. Specifically, the disjunctive graph used to model task relationships in FJSSP aligns well with the DAG-based structure of cloud workflows (addresses **RQ2**). Additionally, the dual-policy PPO design used to coordinate operation selection and machine allocation mirrors the two-stage decision-making structure required in cloud environments, where tasks must be assigned to VMs in an optimal sequence.

While the original model targets scheduling in manufacturing environments with fixed machine pools, this research extends it to cloud computing by modeling workflows as DAGs instead of linear job sequences, using VMs with variable capacities in place of machines, and introducing additional optimization objectives such as energy consumption and QoS-aware latency penalties. These adaptations allow the proposed method to address the complexities of modern cloud environments while building on a robust and scalable reinforcement learning foundation.

5.2 Contributions and Features

This research introduces a novel DRL framework for energy-efficient cloud workflow scheduling, addressing key limitations in existing literature and providing a practical, modular solution. The framework is designed to jointly optimize multiple objectives—makespan, energy consumption, and QoS/SLA adherence—under dynamic, heterogeneous cloud conditions (addresses **RQ1**).

Compared to existing studies, this work contributes a structurally-aware scheduling model that incorporates GNNs to capture task dependencies in DAGs, significantly

improving scheduling adaptability (addresses **RQ2**). The proposed solution supports two-stage scheduling, where high-level and low-level decision policies are coordinated to enable scalable optimization. Additionally, it integrates a reward model based on normalized lower-bound heuristics for energy, makespan, and QoS, improving learning stability and generalization (covers **RQ3**).

A key strength of the proposed system lies in its generalizability. As demonstrated in experiments, models trained on workflows with 20-25 tasks are capable of effectively scaling to scenarios involving up to 1000 tasks. Furthermore, while explicit mechanisms for runtime adaptation are not implemented, the agent implicitly handles dynamic conditions such as varying VM configurations and workflow arrival patterns more effectively than traditional and state-of-the-art baselines.

The current solution is implemented as a modular Python-based scheduler that can interface with CloudSim for simulation-based evaluations. It is designed to be integratable with real-world workflow orchestration platforms such as Apache Airflow or Argo Workflows (addresses **RQ4**). Though not directly deployable in production at present, the architecture supports straightforward API-based extension for integration into cloud-native environments. The deployment artifact development is part of the future work for this solution.

Beyond algorithmic innovation, this research also contributes an integration-oriented design philosophy. Several implementation approaches were explored,including a static DAGFile plugin for Apache Airflow, a Kubernetes-based solution using Kubenetes API Server, and a custom Python-based scheduler with abstracted componenets for DAG scheduling (addresses **RQ4**).

5.3 Experimental Evaluation and Results

To assess the effectiveness of the proposed cloud workflow scheduling framework, a comprehensive evaluation was conducted using both synthetic and real-world datasets under static and dynamic conditions. The model was benchmarked against a range of classical heuristics and state-of-the-art multi-objective evolutionary algorithms. Performance was evaluated using key metrics including Hypervolume, IGD, and Decision Latency. The results collectively validate the framework's ability to optimize multiple objectives, its real-time applicability, and its robustness to dynamic conditions.

On synthetic datasets, the proposed model demonstrated strong performance across all scales of complexity, including 50, 250, and 1000 task workflows. It consistently achieved superior hypervolume and IGD values, with significantly lower decision latency than baseline algorithms. This confirmed its efficiency in generating high-quality trade-offs in near real-time even under increased workload complexity.

In real-world and dynamic scenarios, two agent variants were evaluated: one trained on synthetic data and another fine-tuned on real DAG workloads. Both models

performed competitively, but the agent trained on real workflows achieved the best results. The dynamic variant, which introduced runtime disruptions such as VM failures and inaccurate task duration estimates, revealed the agent's capacity to adapt its scheduling policy without retraining or recomputation. Static heuristic and MOEA-based methods showed notable degradation under the same conditions, confirming the robustness and adaptability of the proposed DRL framework.

Importantly, additional evaluations conducted through workflow-engine and infrastructure-level integrations further demonstrated that performance improvements observed in simulation environments translate into executable systems. Across integration experiments, RL-driven scheduling consistently improved execution efficiency compared to default or heuristic scheduling strategies while preserving compatibility with existing orchestration mechanisms. These findings indicate that the learned policy is not only effective in controlled simulations but also operationally viable within real execution pipelines.

These empirical findings reinforce the conclusion that the proposed framework provides a reliable, scalable, and generalizable solution for energy-efficient and QoS-aware scheduling in heterogeneous, dynamic cloud environments.

5.3.1 Comparative Analysis with Existing Studies

To contextualize the contributions of GNN-Flat and 2SD-GAT, a comparative analysis against several recent studies on cloud workflow scheduling that employ machine learning and reinforcement learning is provided in Table 5.1. The table summarizes key aspects, including optimization objectives, decision models, support for dynamic environments, structural modeling (e.g., DAG awareness), and Pareto-awareness.

5.4 Challenges and Future Work

Despite the strong performance demonstrated by the proposed scheduling framework, several challenges and limitations remain that provide important directions for future research.

A primary challenge concerns computational overhead. The use of graph neural networks combined with deep reinforcement learning, particularly the two-stage decision structure inspired by the 2SD-GAT formulation, increases training complexity and runtime cost compared to traditional heuristic schedulers. While this design substantially reduces the action-space explosion observed in flat DRL formulations, training time grows with workflow size and resource pool scale. For extremely large workflows consisting of thousands of tasks, inference latency may become non-negligible, suggesting the need for hierarchical policies, model compression, or accelerated inference mechanisms.

TABLE 5.1: FEATURE COMPARISON OF RL-BASED SCHEDULING ALGORITHMS

Algorithm/ Method	Workflow Scheduling (DAG)	Dynamic Scenarios	Makespan Objective	Energy Consumption Objective	QoS/SLA Objective	Real Data Inspired Datasets	Pareto-Based Comparisons	Compare with MO Algos	GNNs Utilized	PPO for Training	Multi-Stage/Hierarchical Scheduling
DQTS [15]	✓	✓	✓	✗	✗	✓	✓	✗	✗	✗	✗
DRLB-TSA [16]	✗	✓	✓	✓	✓	✓	✗	✓	✗	✗	✗
DeepRM [23]	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗
DeepRM+ [100]	✗	✓	✗	✗	✗	✓	✗	✗	✗	✗	✗
ADRL [64]	✗	✓	✗	✗	✓	✓	✗	✗	✗	✗	✓
DRL-Cloud [105]	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗	✓
LowCarbon [32]	✓	✓	✗	✓	✓	✓	✗	✗	✗	✗	✗
ODTS [107]	✗	✓	✗	✓	✓	✓	✗	✗	✗	✗	✓
DeepEnergyJSV2.0 [24]	✓	✓	✗	✓	✗	✓	✗	✗	✗	✓	✗
Liu et al. [101]	✗	✓	✗	✓	✓	✓	✗	✗	✗	✗	✓
MARL [3]	✓	✓	✓	✓	✗	✓	✗	✗	✗	✗	✓
IDRQN [103]	✓	✓	✗	✓	✓	✓	✗	✗	✗	✗	✗
DRL + FL [108]	✗	✓	✗	✓	✓	✓	✗	✗	✗	✗	✗
GNN-Flat [141] (Proposed)	✓	✓	✓	✓	✗	✗	✗	✗	✓	✓	✓
2SD-GAT (Proposed)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Another limitation arises from the use of fixed preference-based reward weights for multi-objective optimization. Although training separate agents with different weight configurations enables exploration of Pareto-optimal trade-offs, this approach requires repeated training to cover diverse operational priorities. Future work may investigate preference-conditioned policies, where objective weights are provided as part of the state representation, allowing a single agent to dynamically adapt scheduling behavior according to runtime requirements.

The simulation environment, while effective for controlled evaluation, abstracts several system-level dynamics present in real cloud platforms. Factors such as network

latency variability, virtual machine boot delays, and performance interference from co-located workloads were simplified. Incorporating richer system models would improve realism and help bridge the gap between simulated evaluation and production deployment scenarios.

The current QoS formulation also represents latency using a simplified priority mechanism. Real-world service-level agreements often involve more expressive constraints, including jitter bounds, percentile latency guarantees, and burst deadlines. Extending the framework to support richer QoS representations could enable more accurate modeling of practical cloud service requirements.

Uncertainty modeling remains another open challenge. Real cloud environments exhibit fluctuating resource performance, correlated failures, and partial degradation rather than isolated failures. Integrating uncertainty-aware learning approaches, such as robust reinforcement learning, continual adaptation, or online learning strategies, may improve scheduler stability under non-stationary conditions.

In addition, the current framework assumes a fixed pool of provisioned virtual machines. Extending the scheduler to jointly reason about task placement and dynamic resource provisioning, including elastic scaling and cost-aware instance acquisition, represents an important direction toward production-ready cloud orchestration.

Scalability across geographically distributed and hybrid environments also presents a promising research avenue. Extending the graph representation to capture topology, bandwidth constraints, and data locality could enable effective deployment in edge–cloud and multi-tier computing environments, supporting latency-sensitive applications.

Furthermore, broader benchmarking and real-world validation are needed. Evaluating the framework against a wider range of contemporary DRL scheduling approaches under unified experimental settings, together with longer-running deployments integrated with operational telemetry, would provide deeper insights into generalization capability and practical adoption. Incorporating explainability mechanisms for learned policies may further improve operator trust and facilitate deployment in enterprise cloud systems.

An additional limitation concerns the evaluation scope of the developed event-driven scheduling system and its accompanying web-based graphical interface. While the thesis presents a fully functional prototype capable of real-time scheduling decisions and interactive monitoring, experimental validation was primarily conducted within a controlled simulation setting. Large-scale benchmarking against existing production schedulers using real cloud deployments, multiple virtual machines, and continuously arriving real workloads was not performed. Such evaluation would provide stronger evidence of operational robustness, scalability, and usability under practical conditions, including system overhead, orchestration latency, and user interaction dynamics. Future work will therefore focus on deploying the scheduler within a real-world cloud environment and conducting comparative studies against established scheduling frameworks

using live workloads and heterogeneous infrastructure. This direction would enable assessment of end-to-end system performance beyond algorithmic metrics and support the transition from research prototype to deployment-ready scheduling platform.

CHAPTER 6

CONCLUSION

This thesis addressed the challenges of workflow scheduling in dynamic and heterogeneous cloud environments by proposing a learning-based scheduling framework capable of jointly optimizing makespan, energy consumption, and QoS-aware latency. By combining graph neural networks with deep reinforcement learning, the proposed approach models workflow dependencies explicitly while enabling adaptive decision-making under changing system conditions.

The graph-based formulation allows workflows to be represented as directed acyclic graphs, enabling the scheduler to capture structural relationships among tasks and make dependency-aware scheduling decisions. Through interaction with a simulated cloud environment, the reinforcement learning agent learns policies that adapt at runtime rather than relying on static heuristics or handcrafted rules. The multi-objective design further enables explicit management of trade-offs among execution efficiency, energy usage, and service quality requirements, resulting in stable and competitive scheduling performance across both synthetic and real-world workloads.

Experimental evaluations demonstrate that the proposed framework achieves strong generalization capability and consistent performance under heterogeneous resource conditions. The modular and pluggable implementation also provides practical integration pathways with modern orchestration platforms, helping bridge the gap between simulation-driven research and deployment-oriented cloud scheduling systems.

Beyond the learning framework itself, this thesis also presented the design and implementation of an event-driven scheduling tool prototype accompanied by a web-based graphical user interface. The prototype demonstrates how the proposed scheduler can operate within a practical system setting, supporting real-time event handling, workflow monitoring, and interactive control of scheduling decisions. This system-level implementation highlights the feasibility of integrating learning-based scheduling mechanisms into operational orchestration environments and serves as a bridge between algorithmic research and practical deployment.

Overall, this work shows that integrating graph representation learning with deep reinforcement learning offers an effective and scalable paradigm for dependency-aware workflow scheduling. The results highlight the potential of learning-based schedulers to move beyond static optimization strategies toward adaptive, intelligent resource management mechanisms suitable for next-generation cloud computing environments.

REFERENCES

[1] P. M. Mell and T. Grance, *The NIST definition of cloud computing*. National Institute of Standards and Technology, 2011.

[2] Q. Zhang, L. Cheng, and R. Boutaba, "Cloud computing: state-of-the-art and research challenges," *Journal of internet services and applications*, vol. 1, pp. 7–18, 2010.

[3] A. Jayanetti, S. Halgamuge, and R. Buyya, "Multi-agent deep reinforcement learning framework for renewable energy-aware workflow scheduling on distributed cloud data centers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 35, no. 4, pp. 604–615, 2024.

[4] R. J. Sethi and Y. Gil, "Scientific workflows in data analysis: Bridging expertise across multiple domains," *Future Generation Computer Systems*, vol. 75, pp. 256–270, 2017.

[5] S. Cheikh and J. Walker, "Solving task scheduling problem in the cloud using a hybrid particle swarm optimization approach," *International Journal of Applied Metaheuristic Computing*, vol. 13, pp. 1–25, 2022.

[6] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE transactions on parallel and distributed systems*, vol. 13, no. 3, pp. 260–274, 2002.

[7] J. Malmodin, N. Lövehagen, P. Bergmark, and D. Lundén, "Ict sector electricity consumption and greenhouse gas emissions - 2020 outcome," *Telecommunications Policy*, vol. 48, no. 3, p. 102701, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0308596123002124

[8] G. Zhou, W. Tian, R. Buyya, R. Xue, and L. Song, "Deep reinforcement learning-based methods for resource scheduling in cloud computing: a review and future directions," *Artificial Intelligence Review*, vol. 57, no. 5, 2024.

[9] C. F. Hayes, R. Rădulescu, E. Bargiacchi, J. Källström, M. Macfarlane, M. Reymond, T. Verstraeten, L. M. Zintgraf, R. Dazeley, F. Heintz *et al.*, "A practical guide to multi-objective reinforcement learning and planning," *Autonomous Agents and Multi-Agent Systems*, vol. 36, no. 1, p. 26, 2022.

[10] McKinsey & Company, "How data centers and the energy sector can sate ai's hunger for power," 2024. [Online]. Available: https://www.mckinsey.com/industries/private-capital/our-insights/how-data-centers-and-the-energy-sector-can-sate-ais-hunger-for-power

[11] IEA, "Electricity 2024 - analysis and forecast to 2026," https://iea.blob.core.windows.net/assets/6b2fd954-2017-408e-bf08-952fdd62118a/Electricity2024-Analysisandforecastto2026.pdf, 2024, accessed: 2025-05-17.

[12] S. Di, D. Kondo, and F. Cappello, "Characterizing and modeling cloud applications/jobs on a google data center," *The Journal of Supercomputing*, vol. 69, no. 1, pp. 139–160, 2014.

[13] L. Huang, J. Jia, B. Yu, B.-G. Chun, P. Maniatis, and M. Naik, "Predicting execution time of computer programs using sparse polynomial regression," *Advances in neural information processing systems*, vol. 23, 2010.

[14] E. H. Houssein, A. G. Gad, Y. M. Wazery, and P. N. Suganthan, "Task scheduling in cloud computing based on meta-heuristics: Review, taxonomy, open challenges, and future trends," *Swarm and Evolutionary Computation*, vol. 62, p. 100841, 2021.

[15] Z. Tong, H. Chen, X. Deng, K. Li, and K. Li, "A scheduling scheme in the cloud computing environment using deep q-learning," *Information Sciences*, vol. 512, pp. 1170–1191, 2020.

[16] M. Sudheer, K. Reddy, M. KUMAR, O. Khalaf, C. Romero, and G. Sahib, "DRLBTSA: Deep reinforcement learning based task-scheduling algorithm in cloud computing," *Multimedia Tools and Applications*, vol. 83, pp. 1–29, 2023.

[17] "Apache airflow documentation," 2024. [Online]. Available: https://airflow.apache.org/

[18] "Argo workflows documentation," 2024. [Online]. Available: https://argoproj.github.io/workflows/

[19] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, and R. Buyya, "Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Software: Practice and Experience*, vol. 41, no. 1, pp. 23–50, 2011.

[20] J. J. Durillo, H. M. Fard, and R. Prodan, "Moheft: A multi-objective list-based method for workflow scheduling," in *4th IEEE International Conference on Cloud Computing Technology and Science Proceedings*, 2012, pp. 185–192.

[21] X. Zhou, G. Zhang, J. Sun, J. Zhou, T. Wei, and S. Hu, "Minimizing cost and makespan for workflow scheduling in cloud using fuzzy dominance sort based heft," *Future Generation Computer Systems*, vol. 93, pp. 278–289, 2019.

[22] L. Xing, J. Li, Z. Cai, and F. Hou, "Evolutionary optimization of energy consumption and makespan of workflow execution in clouds," *Mathematics*, vol. 11, no. 9, p. 2126, 2023.

[23] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, "Resource management with deep reinforcement learning," in *Proceedings of the 15th ACM workshop on hot topics in networks*, 2016, pp. 50–56.

[24] Y. Yang, C. He, B. Yin, Z. Wei, and B. Hong, "Cloud task scheduling based on proximal policy optimization algorithm for lowering energy consumption of data center," *KSII Transactions on Internet and Information Systems (TIIS)*, vol. 16, no. 6, pp. 1877–1891, 2022.

[25] K. Lei, P. Guo, Y. Wang, J. Xiong, and W. Zhao, "An end-to-end hierarchical reinforcement learning framework for large-scale dynamic flexible job-shop scheduling problem," in *2022 International Joint Conference on Neural Networks (IJCNN)*, Padua,Italy, 2022, pp. 1–8.

[26] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" *arXiv preprint arXiv:1810.00826*, 2018.

[27] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks," *arXiv preprint arXiv:1710.10903*, 2017.

[28] J. Wilkes, "Google cluster-usage traces v3," https://github.com/google/cluster-data/blob/master/ClusterData2019.md, 2020, accessed: 2025-05-17.

[29] Alibaba, "Alibaba cluster trace program - cluster data v2018," https://github.com/alibaba/clusterdata, 2018, accessed: 2025-05-17.

[30] C. Audet, J. Bigeon, D. Cartier, S. Le Digabel, and L. Salomon, "Performance indicators in multiobjective optimization," *European journal of operational research*, vol. 292, no. 2, pp. 397–422, 2021.

[31] E. Zitzler, K. Deb, and L. Thiele, "Comparison of multiobjective evolutionary algorithms: Empirical results," *Evolutionary computation*, vol. 8, no. 2, pp. 173–195, 2000.

[32] W. Liu, Y. Yan, Y. Sun, H. Mao, M. Cheng, P. Wang, and Z. Ding, "Online job scheduling scheme for low-carbon data center operation: An information and energy nexus perspective," *Applied Energy*, vol. 338, p. 120918, 2023.

[33] K. Lei, P. Guo, Y. Wang, J. Zhang, X. Meng, and L. Qian, "Large-scale dynamic scheduling for flexible job-shop with random arrivals of new jobs by hierarchical

reinforcement learning," *IEEE Transactions on Industrial Informatics*, vol. 20, no. 1, pp. 1007–1018, 2024, conference Name: IEEE Transactions on Industrial Informatics. [Online]. Available: https://ieeexplore.ieee.org/document/10114974

[34] J. Zhou, T. Wang, P. Cong, P. Lu, T. Wei, and M. Chen, "Cost and makespan-aware workflow scheduling in hybrid clouds," *Journal of Systems Architecture*, vol. 100, p. 101631, 2019.

[35] K. M. U. Ahmed, M. H. J. Bollen, and M. Alvarez, "A review of data centers energy consumption and reliability modeling," *IEEE Access*, vol. 9, pp. 152 536–152 563, 2021.

[36] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. J. Maechling, R. Mayani, W. Chen, R. Ferreira da Silva, M. Livny, and K. Wenger, "Pegasus, a workflow management system for science automation," *Future Generation Computer Systems*, vol. 46, pp. 17–35, 2015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X14002015

[37] F. Cao and M. M. Zhu, "Energy-aware workflow job scheduling for green clouds," in *2013 IEEE international conference on green computing and communications and IEEE internet of things and IEEE cyber, physical and social computing*. IEEE, 2013, pp. 232–239.

[38] J. J. Durillo, V. Nae, and R. Prodan, "Multi-objective energy-efficient workflow scheduling using list-based heuristics," *Future Generation Computer Systems*, vol. 36, pp. 221–236, 2014.

[39] R. Shaw, E. Howley, and E. Barrett, "An advanced reinforcement learning approach for energy-aware virtual machine consolidation in cloud data centers," in *2017 12th International Conference for Internet Technology and Secured Transactions (ICITST)*. IEEE, 2017, pp. 61–66.

[40] S. S. Mangalampalli, G. R. Karri, P. R. Ch, K. S. Pokkuluri, P. Chakrabarti, and T. Chakrabarti, "An energy and temperature aware deep reinforcement learning workflow scheduler in cloud computing," *IEEE Access*, vol. 12, pp. 163 424–163 443, 2024.

[41] M. Reymond and A. Nowe, "Pareto-dqn: Approximating the pareto front in complex multi-objective decision problems," in *Proceedings of the Adaptive and Learning Agents Workshop 2019 (ALA-19) at AAMAS*, 2019, pp. 1–6.

[42] W. Zhang and H. Ou, "Reinforcement learning based multi objective task scheduling for energy efficient and cost effective cloud edge computing," *Scientific Reports*, vol. 15, no. 1, 2025.

[43] A. Bartlett, C. Liem, and A. Panichella, "The pursuit of diversity: Multi-objective testing of deep reinforcement learning agents," *arXiv preprint arXiv:2510.14727*, 2025.

[44] Y.-H. Hung, "Investigating how the cloud computing transforms the development of industries," *IEEE Access*, vol. 7, pp. 181 505–181 517, 2019.

[45] M. Ghobakhloo, "The future of manufacturing industry: a strategic roadmap toward industry 4.0," *Journal of manufacturing technology management*, vol. 29, no. 6, pp. 910–936, 2018.

[46] MarketsandMarkets, "Cloud computing market size, size, growth & latest trends," https://www.marketsandmarkets.com/Market-Reports/cloud-computing-market-234.html, 2025, accessed: 2025-05-17.

[47] SPEC, "SPECpower_ssj 2008 results," https://www.spec.org/power_ssj2008/results/power_ssj2008/, 2008, accessed: 2025-05-17.

[48] A. Beloglazov, J. Abawajy, and R. Buyya, "Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing," *Future generation computer systems*, vol. 28, no. 5, pp. 755–768, 2012.

[49] X. Li, W. Yu, R. Ruiz, and J. Zhu, "Energy-aware cloud workflow applications scheduling with geo-distributed data," *IEEE Transactions on Services Computing*, vol. 15, no. 2, pp. 891–903, 2020.

[50] M. Alouane and H. El Bakkali, "Virtualization in cloud computing: Existing solutions and new approach," in *2016 2nd International Conference on Cloud Computing Technologies and Applications (CloudTech)*, 2016, pp. 116–123.

[51] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A {Fault-Tolerant} abstraction for {In-Memory} cluster computing," in *9th USENIX symposium on networked systems design and implementation (NSDI 12)*, 2012, pp. 15–28.

[52] A. Jayanetti, S. Halgamuge, and R. Buyya, "Deep reinforcement learning for energy and time optimized scheduling of precedence-constrained tasks in edge–cloud computing environments," *Future Generation Computer Systems*, vol. 137, pp. 14–30, 2022.

[53] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*. ACM, 2012, pp. 13–16.

[54] D. Kebbal, E. Talbi, and J. Geib, "Building and scheduling parallel adaptive applications in heterogeneous environments," in *ICWC 99. IEEE Computer Society International Workshop on Cluster Computing*. IEEE, 1999, pp. 195–201.

[55] D. Meedeniya, *Deep Learning: A Beginners' Guide*. CRC Press LLC, 2023. [Online]. Available: www.routledge.com/9781032473246

[56] K. Lei, P. Guo, W. Zhao, Y. Wang, L. Qian, X. Meng, and L. Tang, "A multi-action deep reinforcement learning framework for flexible job-shop scheduling problem," *Expert Systems with Applications*, vol. 205, p. 117796, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0957417422010624

[57] Y.-H. Chang, C.-H. Liu, and S. D. You, "Scheduling for the flexible job-shop problem with a dynamic number of machines using deep reinforcement learning," *Information*, vol. 15, no. 2, p. 82, 2024, publisher: Multidisciplinary Digital Publishing Institute. [Online]. Available: https://www.mdpi.com/2078-2489/15/2/82

[58] G. Andreadis, L. Versluis, F. Mastenbroek, and A. Iosup, "A reference architecture for datacenter scheduling: Extended technical report," *arXiv preprint arXiv:1808.04224*, 2018.

[59] S. Euting, C. Janiesch, R. Fischer, S. Tai, and I. Weber, "Scalable business process execution in the cloud," in *2014 IEEE International Conference on Cloud Engineering*. IEEE, 2014, pp. 175–184.

[60] L. Thai, B. Varghese, and A. Barker, "A survey and taxonomy of resource optimisation for executing bag-of-task applications on public clouds," *Future Generation Computer Systems*, vol. 82, pp. 1–11, 2018.

[61] M. Challa and D. Sudha, "An efficient approach for minimization of energy and makespan in cloud computing," *Annals of the Romanian Society for Cell Biology*, vol. 25, no. 6, pp. 7422–7430, 2021.

[62] M. Hussain, L.-F. Wei, A. Lakhan, S. Wali, S. Ali, and A. Hussain, "Energy and performance-efficient task scheduling in heterogeneous virtualized cloud computing," *Sustainable Computing: Informatics and Systems*, vol. 30, p. 100517, 2021.

[63] T. A. Genez, L. F. Bittencourt, and E. R. Madeira, "Workflow scheduling for saas/paas cloud providers considering two sla levels," in *2012 IEEE Network Operations and Management Symposium*, 2012, pp. 906–912.

[64] S. Kardani-Moghaddam, R. Buyya, and K. Ramamohanarao, "Adrl: A hybrid anomaly-aware deep reinforcement learning-based resource scaling in clouds," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 3, pp. 514–526, 2020.

[65] S. Abrishami, M. Naghibzadeh, and D. H. Epema, "Deadline-constrained workflow scheduling algorithms for infrastructure as a service clouds," *Future generation computer systems*, vol. 29, no. 1, pp. 158–169, 2013.

[66] E. Barrett, E. Howley, and J. Duggan, "A learning architecture for scheduling workflow applications in the cloud," in *2011 IEEE ninth European conference on web services*, 2011, pp. 83–90.

[67] W.-N. Chen and J. Zhang, "A set-based discrete pso for cloud workflow scheduling with user-defined qos constraints," in *2012 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, 2012, pp. 773–778.

[68] G. Yao, Y. Ding, and K. Hao, "Using imbalance characteristic for fault-tolerant workflow scheduling in cloud systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 12, pp. 3671–3683, 2017.

[69] X. Zhu, J. Wang, H. Guo, D. Zhu, L. T. Yang, and L. Liu, "Fault-tolerant scheduling for real-time scientific workflows with elastic resource provisioning in virtualized clouds," *IEEE transactions on parallel and distributed systems: a publication of the IEEE Computer Society*, vol. 27, no. 12, pp. 3501–3517, 2016.

[70] X. Ye, J. Li, S. Liu, J. Liang, and Y. Jin, "A hybrid instance-intensive workflow scheduling method in private cloud environment," *Natural Computing*, vol. 18, pp. 735–746, 2019.

[71] M. Masdari, S. ValiKardan, Z. Shahi, and S. I. Azar, "Towards workflow scheduling in cloud computing: a comprehensive analysis," *Journal of Network and Computer Applications*, vol. 66, pp. 64–82, 2016.

[72] A. Verma and S. Kaushal, "Cost minimized pso based workflow scheduling plan for cloud computing," *International Journal of Information Technology and Computer Science*, vol. 7, no. 8, pp. 37–43, 2015.

[73] S. Mohammadi, L. PourKarimi, and H. Pedram, "Integer linear programming-based multi-objective scheduling for scientific workflows in multi-cloud environments," *The Journal of Supercomputing*, vol. 75, no. 10, pp. 6683–6709, 2019.

[74] A. C. Zhou, W. Xue, Y. Xiao, B. He, S. Ibrahim, and R. Cheng, "Taming system dynamics on resource optimization for data processing workflows: A probabilistic approach," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 1, pp. 231–248, 2021.

[75] T. Thanavanich and P. Uthayopas, "Efficient energy aware task scheduling for parallel workflow tasks on hybrids cloud environment," in *2013 International Computer Science and Engineering Conference (ICSEC)*. IEEE, 2013, pp. 37–42.

[76] F. Juarez, J. Ejarque, and R. M. Badia, "Dynamic energy-aware scheduling for parallel task-based application in cloud computing," *Future Generation Computer Systems*, vol. 78, pp. 257–271, 2018.

[77] M. Safari and R. Khorsand, "Energy-aware scheduling algorithm for time-constrained workflow tasks in dvfs-enabled cloud environment," *Simulation Modelling Practice and Theory*, vol. 87, pp. 311–326, 2018.

[78] L. Zhang, L. Wang, Z. Wen, M. Xiao, and J. Man, "Minimizing energy consumption scheduling algorithm of workflows with cost budget constraint on heterogeneous cloud computing systems," *IEEE Access*, vol. 8, pp. 205 099–205 110, 2020.

[79] R. Medara, R. S. Singh, and M. Sompalli, "Energy and cost aware workflow scheduling in clouds with deadline constraint," *Concurrency and Computation: Practice and Experience*, vol. 34, no. 13, p. e6922, 2022.

[80] J. Wang, C. Huang, K. He, X. Wang, X. Chen, and K. Qin, "An energy-aware resource allocation heuristics for vm scheduling in cloud," in *2013 IEEE 10th International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing*. IEEE, 2013, pp. 587–594.

[81] W. Zheng and S. Huang, "An adaptive deadline constrained energy-efficient scheduling heuristic for workflows in clouds," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 18, pp. 5590–5605, 2015.

[82] M. Khaleel and M. M. Zhu, "Energy-efficient task scheduling and consolidation algorithm for workflow jobs in cloud," *International Journal of Computational Science and Engineering*, vol. 13, no. 3, pp. 268–284, 2016.

[83] H. Li, J. Li, W. Yao, S. Nazarian, X. Lin, and Y. Wang, "Fast and energy-aware resource provisioning and task scheduling for cloud systems," in *2017*

18th international symposium on quality electronic design (ISQED), 2017, pp. 174–179.

[84] L. Ye, Y. Xia, S. Tao, C. Yan, R. Gao, and Y. Zhan, "Reliability-aware and energy-efficient workflow scheduling in iaas clouds," *IEEE Transactions on Automation Science and Engineering*, vol. 20, no. 3, pp. 2156–2169, 2022.

[85] N. Mansouri and R. Ghafari, "Cost-efficient task scheduling algorithm for reducing energy consumption and makespan of cloud computing," *Computer and Knowledge Engineering*, vol. 5, no. 1, pp. 1–12, 2022.

[86] S. Pandey, L. Wu, S. M. Guru, and R. Buyya, "A particle swarm optimization-based heuristic for scheduling workflow applications in cloud computing environments," in *2010 24th IEEE international conference on advanced information networking and applications*, 2010, pp. 400–407.

[87] S. Yassa, R. Chelouah, H. Kadima, and B. Granado, "Multi-objective approach for energy-aware workflow scheduling in cloud computing environments," *The Scientific World Journal*, vol. 2013, no. 1, p. 350934, 2013.

[88] K. Sellami, P. F. Tiako, L. Sellami, and R. Kassa, "Energy efficient workflow scheduling of cloud services using chaotic particle swarm optimization," in *2020 IEEE Green Technologies Conference (GreenTech)*. IEEE, 2020, pp. 74–79.

[89] A. A. H. Al-Mahruqi, G. Morison, B. G. Stewart, and V. Athinarayanan, "Hybrid heuristic algorithm for better energy optimization and resource utilization in cloud computing," *Wireless Personal Communications*, vol. 118, no. 1, pp. 43–73, 2021.

[90] N. Khaledian, K. Khamforoosh, R. Akraminejad, L. Abualigah, and D. Javaheri, "An energy-efficient and deadline-aware workflow scheduling algorithm in the fog and cloud environment," *Computing*, vol. 106, no. 1, pp. 109–137, 2024.

[91] B. Xiang, B. Zhang, and L. Zhang, "Greedy-ant: ant colony system-inspired workflow scheduling for heterogeneous computing," *IEEE Access*, vol. 5, pp. 11 404–11 412, 2017.

[92] M. Shojafar, M. Kardgar, A. A. R. Hosseinabadi, S. Shamshirband, and A. Abraham, "Tets: a genetic-based scheduler in cloud computing to decrease energy and makespan," in *Hybrid Intelligent Systems: 15th International Conference HIS 2015 on Hybrid Intelligent Systems, Seoul, South Korea, November 16-18, 2015 15*, 2016, pp. 103–115.

[93] Y. Gu and C. Budati, "Energy-aware workflow scheduling and optimization in clouds using bat algorithm," *Future Generation Computer Systems*, vol. 113, pp. 106–112, 2020.

[94] A. Mohammadzadeh, M. Masdari, and F. S. Gharehchopogh, "Energy and cost-aware workflow scheduling in cloud computing data centers using a multi-objective optimization algorithm," *Journal of Network and Systems Management*, vol. 29, no. 3, p. 31, 2021.

[95] A. Mohammadzadeh, M. Masdari, F. S. Gharehchopogh, and A. Jafarian, "A hybrid multi-objective metaheuristic optimization algorithm for scientific workflow scheduling," *Cluster Computing*, vol. 24, no. 2, pp. 1479–1503, 2021.

[96] A. Mohammadzadeh, M. Akbari Zarkesh, P. Haji Shahmohamd, J. Akhavan, and A. Chhabra, "Energy-aware workflow scheduling in fog computing using a hybrid chaotic algorithm," *The Journal of Supercomputing*, vol. 79, no. 16, pp. 18 569–18 604, 2023.

[97] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: Nsga-ii," *IEEE transactions on evolutionary computation*, vol. 6, no. 2, pp. 182–197, 2002.

[98] K. Deb and H. Jain, "An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part i: Solving problems with box constraints," *IEEE Transactions on Evolutionary Computation*, vol. 18, no. 4, pp. 577–601, 2014.

[99] Q. Zhang and H. Li, "Moea/d: A multiobjective evolutionary algorithm based on decomposition," *IEEE Transactions on evolutionary computation*, vol. 11, no. 6, pp. 712–731, 2007.

[100] W. Guo, W. Tian, Y. Ye, L. Xu, and K. Wu, "Cloud resource scheduling with deep reinforcement learning and imitation learning," *IEEE Internet of Things Journal*, vol. 8, no. 5, pp. 3576–3586, 2021.

[101] N. Liu, Z. Li, J. Xu, Z. Xu, S. Lin, Q. Qiu, J. Tang, and Y. Wang, "A hierarchical framework of cloud resource allocation and power management using deep reinforcement learning," in *2017 IEEE 37th international conference on distributed computing systems (ICDCS)*, 2017, pp. 372–382.

[102] S. Mangalampalli, S. S. Hashmi, A. Gupta, G. R. Karri, K. V. Rajkumar, T. Chakrabarti, P. Chakrabarti, and M. Margala, "Multi objective prioritized workflow scheduling using deep reinforcement based learning in cloud computing," *IEEE access*, vol. 12, pp. 5373–5392, 2024.

[103] H. Lu, C. Gu, F. Luo, W. Ding, and X. Liu, "Optimization of lightweight task offloading strategy for mobile edge computing based on deep reinforcement learning," *Future Generation Computer Systems*, vol. 102, pp. 847–861, 2020.

[104] J. L. Berral, R. Gavalda, and J. Torres, "Adaptive scheduling on power-aware managed data-centers using machine learning," in *2011 IEEE/ACM 12th International Conference on Grid Computing*. IEEE, 2011, pp. 66–73.

[105] M. Cheng, J. Li, and S. Nazarian, "Drl-cloud: Deep reinforcement learning-based resource provisioning and task scheduling for cloud service providers," in *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2018, pp. 129–134.

[106] A. R. Malipatil, M. Paramasivam, D. Gulyamova, A. Saravanan, J. V. N. Ramesh, E. Muniyandy, and R. Ghodhbani, "Energy-efficient cloud computing through reinforcement learning-based workload scheduling." *International Journal of Advanced Computer Science & Applications*, vol. 16, no. 4, 2025.

[107] J. Lou, Z. Tang, and W. Jia, "Energy-efficient joint task assignment and migration in data centers: A deep reinforcement learning approach," *IEEE Transactions on Network and Service Management*, vol. 20, no. 2, pp. 961–973, 2022.

[108] N. Shan, X. Cui, and Z. Gao, ""drl+ fl": An intelligent resource allocation model based on deep reinforcement learning for mobile edge computing," *Computer Communications*, vol. 160, pp. 14–24, 2020.

[109] Z. Zhu, G. Zhang, M. Li, and X. Liu, "Evolutionary multi-objective workflow scheduling in cloud," *IEEE Transactions on parallel and distributed Systems*, vol. 27, no. 5, pp. 1344–1357, 2015.

[110] Y. Zhou and X. Jiao, "Knowledge-driven multi-objective evolutionary scheduling algorithm for cloud workflows," *IEEE Access*, vol. 10, pp. 2952–2962, 2021.

[111] Azure, "Azure public dataset v2," https://github.com/Azure/AzurePublicDataset, 2019, accessed: 2025-05-17.

[112] R. N. Calheiros and R. Buyya, "Meeting deadlines of scientific workflows in public clouds with tasks replication," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 7, pp. 1787–1796, 2013.

[113] W. Hu, X. Li, and X. Li, "Hybrid cloud workflow scheduling method with privacy data," *IEEE Access*, vol. 8, pp. 211 540–211 552, 2020.

[114] A. C. Zhou, B. He, and C. Liu, "Monetary cost optimizations for hosting workflow-as-a-service in iaas clouds," *IEEE transactions on cloud computing*, vol. 4, no. 1, pp. 34–48, 2015.

[115] M. A. Rodriguez and R. Buyya, "Deadline based resource provisioning and scheduling algorithm for scientific workflows on clouds," *IEEE transactions on cloud computing*, vol. 2, no. 2, pp. 222–235, 2014.

[116] Y. Wang and X. Zuo, "An effective cloud workflow scheduling approach combining pso and idle time slot-aware rules," *IEEE/CAA journal of automatica sinica*, vol. 8, no. 5, pp. 1079–1094, 2021.

[117] H. Gu, X. Li, and Z. Lu, "Scheduling spark tasks with data skew and deadline constraints," *IEEE Access*, vol. 9, pp. 2793–2804, 2020.

[118] O. H. Ahmed, J. Lu, A. M. Ahmed, A. M. Rahmani, M. Hosseinzadeh, and M. Masdari, "Scheduling of scientific workflows in multi-fog environments using markov models and a hybrid salp swarm algorithm," *IEEE Access*, vol. 8, pp. 189 404–189 422, 2020.

[119] M. C. Calzarossa, M. L. Della Vedova, L. Massari, G. Nebbione, and D. Tessera, "Multi-objective optimization of deadline and budget-aware workflow scheduling in uncertain clouds," *IEEE Access*, vol. 9, pp. 89 891–89 905, 2021.

[120] N. Zhou, F. Li, K. Xu, and D. Qi, "Concurrent workflow budget-and deadline-constrained scheduling in heterogeneous distributed environments," *Soft Computing*, vol. 22, pp. 7705–7718, 2018.

[121] W. Chen and E. Deelman, "Workflowsim: A toolkit for simulating scientific workflows in distributed environments," in *2012 IEEE 8th international conference on E-science*, 2012, pp. 1–8.

[122] J. J. Durillo and A. J. Nebro, "jmetal: A java framework for multi-objective optimization," *Advances in engineering software*, vol. 42, no. 10, pp. 760–771, 2011.

[123] H. Gupta, A. Vahid Dastjerdi, S. K. Ghosh, and R. Buyya, "ifogsim: A toolkit for modeling and simulation of resource management techniques in the internet of things, edge and fog computing environments," *Software: Practice and Experience*, vol. 47, no. 9, pp. 1275–1296, 2017.

[124] M. C. Silva Filho, R. L. Oliveira, C. C. Monteiro, P. R. Inácio, and M. M. Freire, "Cloudsim plus: a cloud computing simulation framework pursuing software engineering principles for improved modularity, extensibility and correctness,"

in *2017 IFIP/IEEE symposium on integrated network and service management (IM)*, 2017, pp. 400–406.

[125] H. Casanova, "Simgrid: A toolkit for the simulation of application scheduling," in *Proceedings First IEEE/ACM International Symposium on Cluster Computing and the Grid*, 2001, pp. 430–437.

[126] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella, "Multi-resource packing for cluster schedulers," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 455–466, 2014.

[127] "Prefect documentation," 2024. [Online]. Available: https://docs.prefect.io/

[128] "Kubernetes documentation," 2024. [Online]. Available: https://kubernetes.io/

[129] J. Rothman and J. Chamanara, " An RL-Based Model for Optimized Kubernetes Scheduling ," in *2023 IEEE 31st International Conference on Network Protocols (ICNP)*. Los Alamitos, CA, USA: IEEE Computer Society, Oct. 2023, pp. 1–6. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICNP59255.2023.10355623

[130] N. D. Cicco, G. F. Pittalà, G. Davoli, D. Borsatti, W. Cerroni, C. Raffaelli, and M. Tornatore, "Drl-forch: A scalable deep reinforcement learning-based fog computing orchestrator," in *2023 IEEE 9th International Conference on Network Softwarization (NetSoft)*, 2023, pp. 125–133.

[131] R. S. Sutton, "Reinforcement learning: An introduction," *A Bradford Book*, 2018.

[132] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *ArXiv*, vol. abs/1707.06347, 2017. [Online]. Available: https://api.semanticscholar.org/CorpusID:28695052

[133] D. Cordeiro, G. Mounié, S. Perarnau, D. Trystram, J.-M. Vincent, and F. Wagner, "Random graph generation for scheduling simulations," in *3rd International ICST Conference on Simulation Tools and Techniques*, ser. SIMUTools '10. Brussels, Belgium: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2010.

[134] S. Huang, R. F. J. Dossa, C. Ye, J. Braga, D. Chakraborty, K. Mehta, and J. G. Araújo, "Cleanrl: High-quality single-file implementations of deep reinforcement learning algorithms," *Journal of Machine Learning Research*, vol. 23, no. 274, pp. 1–18, 2022.

[135] M. Kumar, S. Sharma, A. Goel, and S. Singh, "A comprehensive survey for scheduling techniques in cloud computing," *Journal of Network and Computer Applications*, vol. 143, pp. 1–33, 2019.

[136] P. ERDdS and A. R&wi, "On random graphs i," *Publ. math. debrecen*, vol. 6, no. 290-297, p. 18, 1959.

[137] W. M. van Der Aalst, A. H. Ter Hofstede, B. Kiepuszewski, and A. P. Barros, "Workflow patterns," *Distributed and parallel databases*, vol. 14, pp. 5–51, 2003.

[138] S. Fu, "Failure-aware resource management for high-availability computing clusters with distributed virtual machines," *Journal of Parallel and Distributed Computing*, vol. 70, no. 4, pp. 384–393, 2010.

[139] "Kind documentation," 2026. [Online]. Available: https://kind.sigs.k8s.io/

[140] M. Li and X. Yao, "Quality evaluation of solution sets in multiobjective optimisation: A survey," *ACM Computing Surveys*, vol. 52, pp. 1–38, 03 2019.

[141] S. Chandrasiri and D. Meedeniya, "Energy-efficient dynamic workflow scheduling in cloud environments using deep learning," *Sensors*, vol. 25, no. 5, p. 1428, 2025.

[142] ——, "Multi-objective cloud workflow scheduling with two-stage deep reinforcement learning and graph neural networks," *IEEE Access*, 2026.

APPENDIX A

PUBLICATIONS

1. S. Chandrasiri, D. Meedeniya, *"Energy-Efficient Dynamic Workflow Scheduling in Cloud Environments Using Deep Learning"*, Sensors, vol. 25, no. 5: 1428, MDPI, 2025 [141].
   DOI: 10.3390/s25051428
   Journal Rank: Q1
   H-Index: 273
   ISSN / eISSN: 1424-8220
   Indexed in: Scopus, Web of Science Core Collection (Science Citation Index Expanded), Current Contents - Engineering, Computing & Technology, Essential Science Indicators

2. S. Chandrasiri, D. Meedeniya, *Multi-Objective Cloud Workflow Scheduling with Two-Stage Deep Reinforcement Learning and Graph Neural Networks*, IEEE Access, 2026 [142].
   DOI: 10.1109/ACCESS.2026.3669772
   Journal Rank: Q1
   H-Index: 290
   ISSN / eISSN: 2169-3536
   Indexed in: Scopus, Web of Science Core Collection (Science Citation Index Expanded), Current Contents - Engineering, Computing & Technology, Essential Science Indicators

APPENDIX B

2SD-GAT AND TOOL USER MANUAL

This Command Line Interface (CLI) trains a reinforcement learning scheduling agent using the proposed 2SD-GAT method (Section 3.3).

B.1 Setting up the Environment

Before running the training script, ensure that all required dependencies are installed. The codebase relies on Python 3.11+. The environment can be set up using the provided `requirements.txt` file:

Listing B.1: Installing dependencies.

```
pip install -r requirements.txt
```

This will install all necessary Python packages, including PyTorch, PyTorch Geometric, and Gymnasium.

B.2 Training the Agent

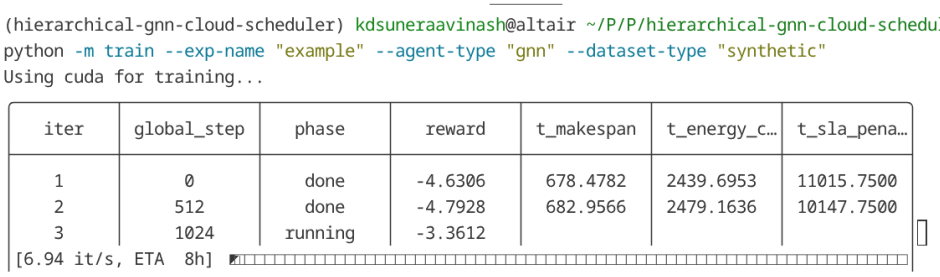

Fig. B.1: Training script execution

The training script `train.py` is the main entry point for training 2SD-GAT agents. The script performs the full experimental pipeline, including dataset generation, environment creation, agent initialization, PPO training, periodic evaluation, and checkpointing. All experiment parameters are configurable through command-line arguments.

Run training with default parameters:

Listing B.2: Running training with default configuration.

```
python -m train
```

Each execution creates a new run directory in the `logs` folder, named with a timestamp and optional experiment name with format `<timestamp>_<exp_name>`. To specify an experiment name:

Listing B.3: Running an experiment with a custom name.

```
python -m train --exp-name experiment1
```

The CLI accepts a wide range of parameters to customize the training process (Table B.1). These parameters control various aspects of the experiment, including optimization settings, dataset generation, reward preferences, and logging configurations. For example, to change the learning rate and number of parallel environments:

Listing B.4: Changing learning rate and parallel environments.

```
python -m train --learning-rate 0.0001 --num-envs 8
```

The neural architecture is chosen using the following command-line argument.Here, the available agent types are `mlp` for a simple feedforward network, `gin` for a GIN-based architecture, `flat-gnn` for a GIN-based architecture with a flat action space, and `gnn` for the proposed 2SD-GAT architecture.

Listing B.5: Selecting the agent architecture.

```
python -m train --agent-type gnn
```

Training length is controlled by total environment steps which is specified via the `-total-timesteps` argument. The default is set to 200,000 steps.

It is possible to select between different dataset generation strategies using the `-dataset-type` argument. The default is `synthetic` which generates random task sets (DS_i^{syn}) and machine configurations. `real_world` generates datasets based on real-world traces ((DS^{real})), and `real_world_dynamic` creates real-world datasets with dynamic properties (DS^{dyn}).

After each training iteration, the agent is evaluated on fixed-seed datasets. Makespan, energy consumption, and QoS penalty values are logged for each evaluation episode. These metrics are synchronized automatically with TensorBoard. To launch visualization:

Listing B.6: Launching TensorBoard.

```
tensorboard --logdir logs
```

Models are saved periodically at checkpoints during training, with filenames indicating the training step and the latest checkpoint always saved as `model.pt`. For example, after 10,000 steps, a checkpoint named `model_10000.pt` will be created. At the end of training, the final model will be saved as `model.pt` in the same directory. A checkpoint is created approximately every 10,000 steps.

TABLE B.1: CLI CONFIGURATION PARAMETERS FOR PPO TRAINING

Parameter	Description
Experiment Configuration	
`exp-name`	Name of the experiment run
`agent-type`	Type of DRL agent architecture used for training
`seed`	Random seed for reproducibility
`output-dir`	Directory where logs and checkpoints are stored
`torch-deterministic`	Enables deterministic CUDA operations
`cuda`	Enables GPU acceleration if available
`track`	Enables experiment tracking with Weights & Biases
`wandb-project-name`	WandB project name used for tracking
`wandb-entity`	WandB team or user entity
`load-model-dir`	Directory containing a pretrained model to load
`test-iterations`	Number of evaluation runs during testing phase
PPO Training Parameters	
`total-timesteps`	Total number of environment interaction steps
`learning-rate`	Learning rate of Adam optimizer
`num-envs`	Number of parallel environments
`num-steps`	Steps collected per rollout per environment
`anneal-lr`	Enables linear learning-rate annealing
`gamma`	Discount factor for future rewards
`gae-lambda`	GAE lambda parameter for advantage estimation
`num-minibatches`	Number of minibatches per PPO update
`update-epochs`	Number of optimization epochs per rollout
`norm-adv`	Normalizes computed advantages
`clip-coef`	PPO surrogate objective clipping coefficient
`clip-vloss`	Uses clipped value loss
`ent-coef`	Entropy bonus coefficient encouraging exploration
`vf-coef`	Value function loss coefficient
`max-grad-norm`	Maximum gradient norm for clipping
`target-kl`	Optional KL divergence threshold for early stopping
Dataset and Reward Configuration	
`dataset-type`	Dataset generation strategy
`makespan-pref`	Reward weight for minimizing makespan
`energy-pref`	Reward weight for minimizing energy consumption
`sla-penalty-pref`	Reward weight for minimizing QoS penalties
`makespan-alpha`	Multiplier for the makespan reward
`energy-alpha`	Multiplier for the energy consumption reward
`sla-penalty-alpha`	Multiplier for the QoS penalty reward

B.3 Tool Execution via CLI

As introduced in Section 3.7, the developed scheduling tool consists of three main components: the Engine, the Workers, and the Submission Script. The system follows an

event-driven architecture where all components communicate asynchronously through a RabbitMQ message broker. Each component connects to the same broker instance to exchange messages and coordinate scheduling operations.

```
(pluggable-scheduler) kdsuneraavinash@altair ~/P/P/pluggable-scheduler (master)> uv run main.py config:auto-run --config.rabbitmq-url amqp://guest:guest@lo
calhost:5672/ --config.input-file sample.json --config.algorithm ferpts
2026-03-02 02:20:38.604 | INFO     | models.events:parse:23 - Parsing event from raw data: {"type":"task_completion","task_id":{"value":"3_42"},"energy_con
sumed":4363.754666113646,"result":"OK"}
2026-03-02 02:20:38.604 | INFO     | engine.core:handle_task_completion:83 - Task completed: value='3_42' with result OK and energy 4363.754666113646
2026-03-02 02:20:38.791 | INFO     | engine.core:trigger_scheduling:128 - Triggering scheduling process
2026-03-02 02:20:38.791 | INFO     | engine.strategies.ferpts:schedule:21 - Scheduling with FERPTS with 11 workers and 250 tasks
2026-03-02 02:20:38.793 | INFO     | engine.core:trigger_scheduling:133 - Scheduling result: (WorkerId(value='vm-5'), TaskId(value='6_90'))
2026-03-02 02:20:38.793 | INFO     | engine.core:emit:51 - Emitting command: type='start_task' worker_id=WorkerId(value='vm-5') task_id=TaskId(value='6_90'
) task_definition=TaskDefinition(id=TaskId(value='6_90'), command="echo 'NOP'", priority=0, dependencies=[TaskId(value='6_86')], estimated_length_mi=104592
, req_memory_gb=9.0, req_core_count=8.0)
2026-03-02 02:20:38.793 | DEBUG    | connector.rabbitmq:connect:29 - Already connected to worker value='vm-5'
2026-03-02 02:20:38.794 | DEBUG    | connector.rabbitmq:send_to_worker:40 - Sent message to worker value='vm-5'
2026-03-02 02:20:39.904 | INFO     | engine.core:trigger_scheduling:128 - Triggering scheduling process
2026-03-02 02:20:39.904 | INFO     | engine.strategies.ferpts:schedule:21 - Scheduling with FERPTS with 11 workers and 250 tasks
2026-03-02 02:20:39.906 | INFO     | engine.core:trigger_scheduling:133 - Scheduling result: (WorkerId(value='vm-5'), TaskId(value='6_91'))
2026-03-02 02:20:39.906 | INFO     | engine.core:emit:51 - Emitting command: type='start_task' worker_id=WorkerId(value='vm-5') task_id=TaskId(value='6_91'
) task_definition=TaskDefinition(id=TaskId(value='6_91'), command="echo 'NOP'", priority=0, dependencies=[TaskId(value='6_86')], estimated_length_mi=109728
, req_memory_gb=9.0, req_core_count=4.0)
2026-03-02 02:20:39.906 | DEBUG    | connector.rabbitmq:connect:29 - Already connected to worker value='vm-5'
2026-03-02 02:20:39.907 | DEBUG    | connector.rabbitmq:send_to_worker:40 - Sent message to worker value='vm-5'
2026-03-02 02:20:41.024 | INFO     | engine.core:trigger_scheduling:128 - Triggering scheduling process
2026-03-02 02:20:41.024 | INFO     | engine.strategies.ferpts:schedule:21 - Scheduling with FERPTS with 11 workers and 250 tasks
2026-03-02 02:20:41.026 | INFO     | engine.core:trigger_scheduling:133 - Scheduling result: (WorkerId(value='vm-8'), TaskId(value='6_87'))
2026-03-02 02:20:41.026 | INFO     | engine.core:emit:51 - Emitting command: type='start_task' worker_id=WorkerId(value='vm-8') task_id=TaskId(value='6_87'
) task_definition=TaskDefinition(id=TaskId(value='6_87'), command="echo 'NOP'", priority=0, dependencies=[TaskId(value='6_86')], estimated_length_mi=97572,
 req_memory_gb=4.0, req_core_count=9.0)
2026-03-02 02:20:41.026 | DEBUG    | connector.rabbitmq:connect:29 - Already connected to worker value='vm-8'
2026-03-02 02:20:41.027 | DEBUG    | connector.rabbitmq:send_to_worker:40 - Sent message to worker value='vm-8'
2026-03-02 02:20:41.027 | DEBUG    | connector.rabbitmq:consume_from_engine:66 - Received message from engine for worker value='vm-8' and processing it
2026-03-02 02:20:41.027 | INFO     | worker.core:handle_start_task:37 - Starting task value='6_87' with definition id=TaskId(value='6_87') command="echo 'N
OP'" priority=0 dependencies=[TaskId(value='6_86')] estimated_length_mi=97572 req_memory_gb=4.0 req_core_count=9.0
2026-03-02 02:20:41.027 | INFO     | worker.core:handle_start_task:41 - Mock executing task value='6_87' for 10.64 seconds
2026-03-02 02:20:42.141 | INFO     | engine.core:trigger_scheduling:128 - Triggering scheduling process
2026-03-02 02:20:42.141 | INFO     | engine.strategies.ferpts:schedule:21 - Scheduling with FERPTS with 11 workers and 250 tasks
2026-03-02 02:20:42.143 | INFO     | engine.core:trigger_scheduling:133 - Scheduling result: (WorkerId(value='vm-5'), TaskId(value='3_43'))
```

Fig. B.2: Tool execution via CLI

The engine acts as the central coordinator responsible for managing scheduling requests and maintaining the global system state. The engine must be executed on a single server instance to prevent coordination conflicts.

Listing B.7: Running the scheduling engine.

```
python -m main config:engine \
    --config.rabbitmq-url <rabbitmq-url>
```

The parameter `<rabbitmq-url>` specifies the connection string to the RabbitMQ broker, for example `amqp://user:password@host:5672/`. Only one engine instance should be active at any given time.

Workers are responsible for performing scheduling computations. Multiple workers can be deployed across different machines to increase computational capacity and improve throughput. Each worker connects to the message broker and receives tasks dynamically from the engine.

Listing B.8: Running a scheduling worker.

```
python -m main config:worker \
        --config.rabbitmq-url <rabbitmq-url> \
        --config.worker-id <worker-id>
```

Each worker must be assigned a unique identifier through `<worker-id>`. Running multiple workers connected to the same RabbitMQ instance creates a distributed pool capable of parallel processing.

Workflow submissions are performed using the submission component, which publishes scheduling requests to the engine through the message broker.

Listing B.9: Running the submission script.

```
python -m main config:submit \
    --config.rabbitmq-url <rabbitmq-url> \
    --config.workflow-file <workflow-file-path>
```

The workflow definition must be provided as a JSON file describing workflows and tasks.

Listing B.10: Example workflow file format.

```
{
  "workflows": [
    {
        "id": 0,
        "arrival_time": 0,
        "priority": 0
    }
  ],
  "tasks": [
    {
      "id": 0,
      "workflow_id": 0,
      "child_ids": [1, 2, 3, 4],
      "command": "python task.py",
      "length": 137458,
      "req_memory_gb": 9,
      "req_core_count": 4
    }
  ]
}
```

B.4 Tool Execution via GUI

Figure B.3 presents the web-based graphical user interface developed for workflow submission and scheduling monitoring. The interface provides an integrated environment for defining workflows, configuring resources, validating inputs, and observing

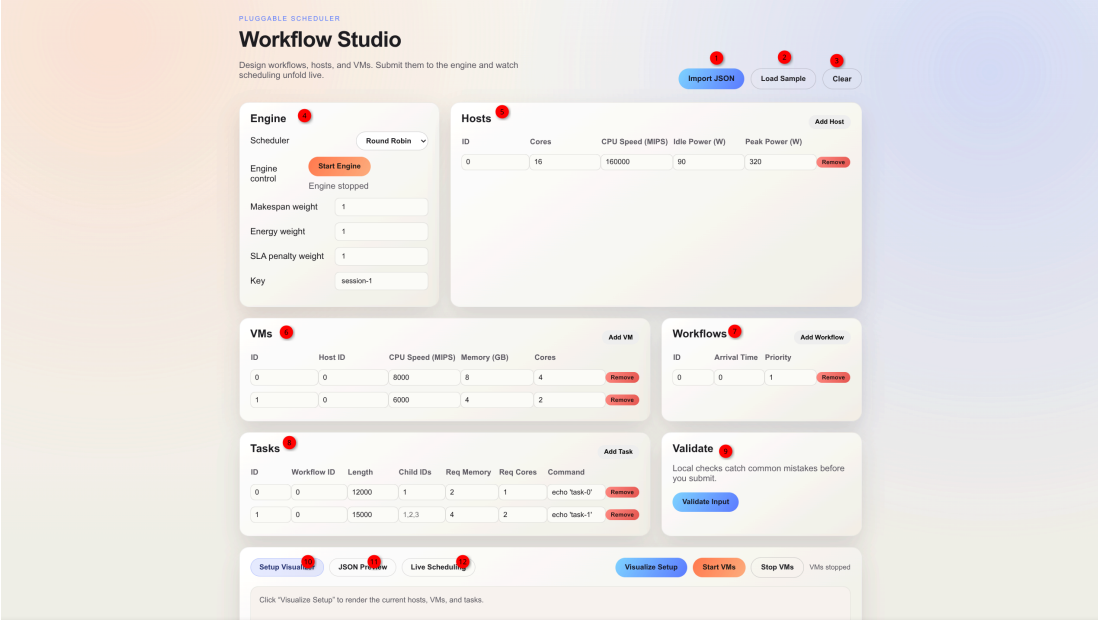

Fig. B.3: Screenshot of the scheduling GUI, with numbered elements corresponding to different functionalities

scheduling progress in real time. The numbered elements highlighted in the figure correspond to the functionalities described in Table B.2.

TABLE B.2: FUNCTIONAL ELEMENTS OF THE SCHEDULING GUI

No.	Functionality Description
1	Button used to select and open an exported workflow definition in JSON format.
2	Button for loading a predefined sample workflow definition.
3	Button to clear the currently loaded workflow definition from the interface.
4	Panel containing scheduling engine configuration options.
5	Panel for defining physical host configurations.
6	Panel for configuring virtual machines or worker resources.
7	Panel for defining workflow-level parameters.
8	Panel for specifying task properties and dependency relationships.
9	Button to validate workflow definitions and configuration settings.
10	Button to submit the workflow to the scheduling engine.
11	Visualization panel displaying the configured setup.
12	Panel for previewing and exporting the generated JSON workflow definition.
13	Panel displaying real-time scheduling progress using a Gantt chart.

The GUI can be launched using the following command:

Listing B.11: Running the scheduling GUI.

```
python -m admin.web_gui \
    --config.rabbitmq-url <rabbitmq-url>
```

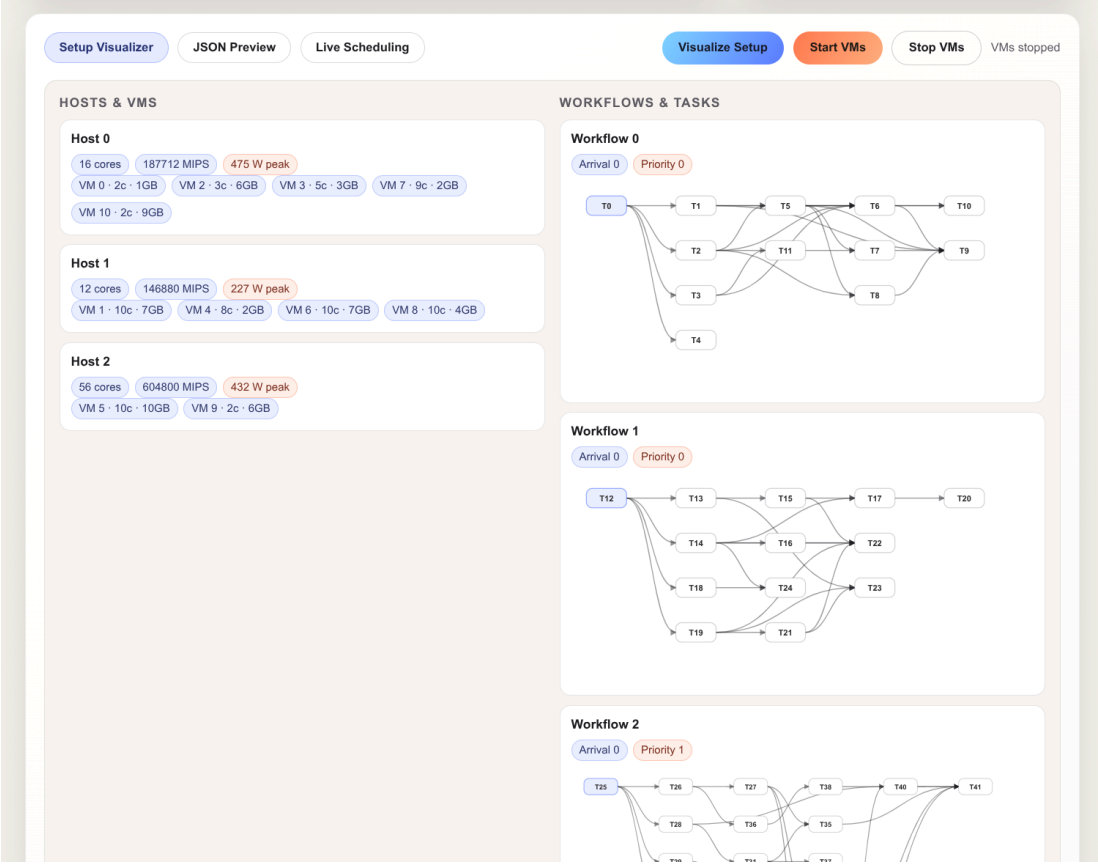

Fig. B.4: Screenshot of the VM/workflow visualizer in the web GUI, showing the defined hosts, VMs, and task DAGs

The interface connects to the same RabbitMQ instance used by the engine and worker components, enabling workflow submission and continuous reception of scheduling updates. In addition to submission capabilities, the GUI supports simulation by allowing users to define host configurations, virtual machines, and workflow structures prior to execution.

Within the Engine panel, users may select the scheduling strategy to be applied. Available schedulers include the reinforcement learning–based scheduler proposed in this thesis as well as heuristic baselines such as Round-Robin, HEFT, and FERPTS.

The Host and VM panels allow users to specify cluster characteristics, including computational capacity and resource allocation. These configurations are primarily intended for simulation scenarios. When workflows are submitted to a real cluster, these fields may remain empty because the live cluster state is automatically used during scheduling.

The Workflow and Task panels enable detailed workflow construction. Users can define workflow metadata, task dependencies, and resource requirements, thereby constructing the DAG representing execution order.

The visualization panel provides a graphical representation of the configured envi-

ronment and workflow structure. This visualization assists users in verifying correctness prior to submission by displaying hosts, virtual machines, and workflow DAG relationships.

The Engine panel also includes a *Start Engine* option that launches a local engine instance directly from the GUI. This functionality is intended for experimentation and simulation. For production deployments or real cluster executions, the engine should be started independently on the target infrastructure. Similarly, the setup visualizer enables users to create hosts and virtual machines interactively for simulation purposes, whereas real deployments rely on existing cluster resources.

The JSON preview panel allows users to inspect the generated workflow specification before submission. The workflow definition may also be exported as a JSON file for later reuse or for submission via the CLI. Once submitted, the JSON specification is transmitted to the engine through RabbitMQ, initiating the scheduling process.

Fig. B.5: Screenshot of the scheduling status visualizer in the web GUI, showing a Gantt chart of task execution

Scheduling progress is visualized in real time within the Gantt chart panel, where task execution timelines are displayed as the schedule is produced and executed.

APPENDIX C

CODE & DATA AVAILABILITY

1. The code for the GNN-Flat scheduler and related components is available at:
   https://github.com/kdsuneraavinash/workflow-cloudsim-drlgnn

2. The code for the 2SD-GAT scheduler and related components, including integrations with Apache Airflow and Kubernetes, is available at:
   https://github.com/kdsuneraavinash/2sd-gat-cloud-scheduler

3. The code for the pluggable RL-based scheduling framework (CLI + GUI) is available at:
   https://github.com/kdsuneraavinash/prototype-workflow-engine

4. The Google Cloud Trace dataset used in this research is available at:
   https://github.com/google/cluster-data

5. The real world host characteristics dataset (SPECpower_ssj 2008 Results) used in this research is available at:
   https://www.spec.org/power_ssj2008/results/