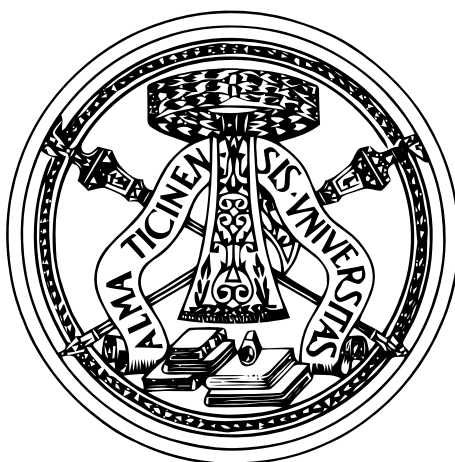




UNIVERSITY OF PAVIA

FACULTY OF ENGINEERING

DEPARTMENT OF ELECTRICAL, COMPUTER AND BIOMEDICAL ENGINEERING

PHD IN ELECTRONICS, COMPUTER SCIENCE AND ELECTRICAL ENGINEERING

XXXVIII CYCLE

PHD THESIS

***Techniques and Tools for Resource Management in
Cloud-Edge Environments***

Candidate: Luca Zanussi

Supervisor: Prof. Maria Carla Calzarossa

Co-Supervisor: Prof. Luisa Massari

Academic Year 2024/2025

ABSTRACT

Cloud and Cloud-Edge platforms provide distributed computing resources to execute a wide range of workloads. Exploiting these platforms effectively requires careful management of how resources are provisioned and how workloads are mapped to the allocated resources. Two complementary resource management problems are addressed in this thesis.

The first research problem addresses the scheduling of batch workloads composed of multiple workflows represented as directed acyclic graphs. The resources to be provisioned refer to a Cloud-Edge infrastructure. These resources are characterized by performance uncertainty and variability. The problem is formulated as a probabilistic multi-objective optimization problem aimed at minimizing the percentiles of the probability distributions of the workload execution time and resource monetary cost. The framework designed and developed for solving this problem combines a customized Genetic Algorithm and a Monte Carlo simulation that estimates the probability distributions. Simulation results on synthetic and real workloads show that the framework is able to identify a wide range of scheduling plans which provide optimal tradeoffs between execution time and monetary cost.

The second research problem addresses the autoscaling of Cloud resources for interactive workloads consisting of streams of requests with time-varying arrival rates. A Load-Aware Predictive framework is designed and developed to adjust the allocated resources by anticipating future demands. The framework is organized as a closed control loop invoked at every control cycle and built around two main components. A workload predictor produces a short-horizon forecast of the arrival rate from its recent measurements. A capacity estimator then derives the number of resources required to sustain a target utilization. Since utilization and the resource count depend on the throughput of the resources and steady-state conditions cannot be assumed under varying load, the throughput is not equal to the arrival rate and must be estimated: requests that the allocated resources cannot serve accumulate as a backlog of unprocessed requests, which the framework computes by propagating it across the horizon to account for the accumulated demand. Scale-out actions are issued promptly to absorb rising demand, whereas scale-in actions are conservative and triggered only under persistent over-provisioning.

Simulation results on synthetic and real workloads show that the framework copes very well with fluctuating workloads and is more robust than state of the art policies.

CONTENTS

1	Introduction	1
2	Foundational Concepts	5
2.1	Workload Models	6
2.1.1	Batch Workload	6
2.1.2	Interactive Workload	7
2.2	Cloud-Edge Continuum	7
2.3	Workflow Scheduling	9
2.3.1	Problem Classification	9
2.3.2	Solution Approaches	10
2.4	Resource Provisioning with Autoscaling	12
2.4.1	Conceptual Model	13
2.4.2	Scaling Actions	14
2.4.3	Control Strategies	15
3	State of the Art	16
3.1	Workflow Scheduling in the Cloud-Edge Continuum	17

3.1.1	Workflow scheduling in Cloud environments	17
3.1.2	Workflow scheduling in the Cloud-Edge Continuum . . .	24
3.1.3	Multiple Workflow Scheduling	27
3.1.4	Scheduling under uncertainty	32
3.2	Autoscaling Techniques in Cloud Environments	38
3.2.1	Reactive approaches	38
3.2.2	Proactive approaches	42
3.2.3	Hybrid approaches	52
4	Problem Definition and Methodology	58
4.1	Models for Workflow Scheduling	59
4.1.1	Workload Model	59
4.1.2	Cloud-Edge Model	61
4.1.3	Execution and Cost Models	61
4.2	Multi-Objective Workflow Scheduling Problem	64
4.2.1	Problem Formulation	64
4.2.2	Problem Structure	66
4.2.3	Solution Approach	67
4.3	Autoscaling Strategy	67
4.3.1	Workload and Infrastructure Models	67
4.3.2	Capacity Management	68
4.4	Load-Aware Predictive Autoscaling Strategy	69
4.5	Autoscaling with Heterogeneous Resources	71

5	Workflow Scheduling Framework	73
5.1	Scheduling Plan Encoding	74
5.2	Evolutionary Process	75
5.2.1	Crossover Operators	75
5.2.2	Mutation Operators	76
5.3	Scheduling Plan Evaluation	77
5.4	Initial Population	78
6	Experimental Evaluation of Workflow Scheduling	81
6.1	Experimental Setup	82
6.1.1	Workloads	82
6.1.2	Cloud-Edge Infrastructure	87
6.1.3	Simulation Settings	88
6.2	Results	90
6.2.1	Benefit of the Cloud-Edge Continuum	90
6.2.2	Behavior on Multiple Workflows	92
6.2.3	Impact of Sequential versus Multiple Workflow Execution	95
6.2.4	Impact of Mixed Workflows	96
6.2.5	Effect of the Initial Population	97
6.2.6	Impact of Performance Variability	99
6.2.7	Impact of Deadline Constraints	100
7	Autoscaling Framework	103
7.1	Workload Predictor	104

7.2	Resource Selector	106
7.2.1	Scale-Out policy	106
7.2.2	Scale-In policy	108
7.3	Heterogeneous Resource Selection	110
8	Experimental Evaluation of Autoscaling	112
8.1	Experimental Setup	113
8.1.1	Workloads	113
8.1.2	Cloud Infrastructure	115
8.1.3	Framework and Simulation Settings	116
8.2	Results	118
8.2.1	Comparison with Baselines on Synthetic Workloads . . .	119
8.2.2	Comparison with Baselines on the Flight Seats Workload	121
8.2.3	Impact of the Desired Utilization Level	121
8.2.4	Impact of Heterogeneous Resources	122
8.2.5	Comparison of Heterogeneous and Homogeneous Resources	125
9	Conclusion and Future Work	127

LIST OF FIGURES

2.1	Example of a workflow represented as a Directed Acyclic Graph.	6
2.2	Layered architecture of the Cloud-Edge Continuum.	8
2.3	Representation of a MAPE-K control loop.	14
4.1	Example of a workload composed of two independent work- flows, each represented as a DAG.	59
4.2	Overview of the proposed scheduling framework.	65
4.3	Overview of the proposed autoscaling framework.	70
5.1	Chromosome encoding of a scheduling plan for an example workflow.	74
5.2	Crossover operators applied to scheduling and provisioning chro- mosomes.	76
5.3	Mutation operators applied to scheduling and provisioning chro- mosomes.	76
6.1	Structure of the IoT data processing workflow.	82

6.2	Structure of the artificially generated workflow used to derive the synthetic workloads.	83
6.3	Structure of a representative Cycles workflow.	85
6.4	Structure of a representative CyberShake workflow.	86
6.5	Pareto fronts for the balanced IoT data processing workflow as a function of the resources being provisioned.	90
6.6	Pareto fronts for the synthetic workloads under Cloud-only and Edge-only infrastructures.	91
6.7	Pareto fronts of the proposed strategy and MOHEFT for the 12 synthetic workloads.	93
6.8	Number of Cloud and Edge resources allocated by the optimal scheduling plans for the 12 synthetic workloads.	94
6.9	Execution times of four sequential standalone workflows compared to those of the same four workflows interleaved in a single workload.	95
6.10	Pareto fronts for the four mixed-workflow synthetic workloads. .	96
6.11	Number of Cloud and Edge resources allocated by the optimal scheduling plans for the mixed-workflow workloads.	97
6.12	Pareto fronts obtained with the customized and a randomly generated initial population.	98
6.13	Pareto fronts for the computation-intensive IoT data processing workflow as a function of the variability in processing capacity and network bandwidth.	99

6.14 Pareto fronts for the Cycles and CyberShake workloads as a function of the variability factor.	100
6.15 Pareto fronts for the Cycles workload under three deadlines and four variability factors.	101
6.16 Pareto fronts for the CyberShake workload obtained by constraining one workflow at a time, for three variability factors.	102
7.1 Example of a polynomial model fitted to a window of arrival rates, with forecasts beyond the fitting window.	104
7.2 Rolling window update of the workload predictor.	105
8.1 Arrival patterns of the synthetic workloads.	113
8.2 Arrival patterns of the real workloads.	114
8.3 Architecture of the autoscaling simulation environment.	117
8.4 Workload arrival rate and number of VMs allocated by the Load-Aware Predictive and Average Load policies on the synthetic workloads.	120
8.5 Number of allocated VMs over 24 hours by varying the desired utilization level on the flight seats availability workload.	122
8.6 Response time as a function of the total VM leasing time for different values of the desired utilization level on the flight seats availability workload.	123
8.7 Breakdown of the number of VMs allocated by the autoscaler on the Redset workload.	124
8.8 Average utilization of the allocated VMs on the Redset workload.	125

LIST OF TABLES

4.1	Summary of the main notation used in the context of workflow scheduling.	60
4.2	Summary of the main notation used in the context of autoscaling.	68
6.1	Overall processing demands and data volumes of the four mixed-workflow synthetic workloads.	84
6.2	Overall processing demands and data volumes of the three Cycles workflows.	86
6.3	Overall processing demands and data volumes of the three CyberShake workflows.	87
6.4	Characteristics of the Cloud-Edge resources considered in the experiments.	87
6.5	Front extremes with the customized and random initial population.	98
6.6	Cloud and Edge resource counts for three scheduling plans of the Cycles workload.	101

8.1	Characteristics of the synthetic workloads considered in the experiments.	114
8.2	Characteristics of the heterogeneous VM catalog used in the experiments.	116
8.3	Autoscaler and simulation parameters.	118
8.4	Comparison of the performance of the Load-Aware Predictive policy and of the three baselines on the synthetic workloads. . .	119
8.5	Comparison of the performance of the Load-Aware Predictive policy against the Average Load baseline on the flight seats availability workload.	121
8.6	Response time distribution on the flight seats availability workload.	123
8.7	Per-type allocation statistics on the Redset workload under the heterogeneous configuration.	124
8.8	Heterogeneous against single-type configurations on the Redset workload.	126

CHAPTER 1

INTRODUCTION

Cloud computing has become the dominant paradigm for provisioning computational resources on demand thanks to its flexibility, scalability, and cost effectiveness. By offering virtualized infrastructure with a pay per use pricing model, Cloud platforms allow companies to acquire processing capacity and storage without upfront investment in physical hardware. As the strategic importance of Cloud computing keeps growing, Gartner [1] predicts that by 2028 it will no longer be just a disruptive technology but a core necessity for sustaining competitive advantage.

At the same time, application requirements have evolved, with Cloud infrastructure being complemented by computing capabilities placed closer to end users and data sources. The result is the Cloud-Edge Continuum, which integrates heterogeneous resources spanning from Cloud data centers to the Edge of the network to, e.g., mobile and Internet of Things (IoT) devices. Across this infrastructure, resource management is a critical concern, as the allocation of resources to applications directly affects their performance and monetary cost.

This thesis studies resource management for two classes of workloads, i.e., batch and interactive workloads. Batch workloads consist of applications, each structured as a set of interdependent tasks. These applications are commonly represented as workflows characterized by control and data dependencies. In contrast, interactive workloads consist of streams of requests submitted by users to access a service, whose arrival rate varies over time and is not known in advance.

In the context of batch workloads, resource management decisions have to identify the resources to be provisioned and the mapping of workflow tasks onto them that satisfies task precedence constraints.

These decisions are challenging for three reasons. First, the number of possible mappings grows combinatorially with the number of tasks and resources, making exact methods unsuitable for problems of practical interest. Second, the Cloud-Edge Continuum offers highly heterogeneous resources, thus requiring to choose the "best" tradeoff between performance and cost. Third, co-location and virtualization techniques commonly employed by Cloud and Edge platforms introduce resource performance uncertainty, thus requiring explicit models of these behaviors.

Existing scheduling methods largely target single workflows in Cloud-only settings and typically assume deterministic task execution times, leaving the joint minimization of execution time and cost for multiple workflows under performance uncertainty largely unaddressed across the Cloud-Edge Continuum. These limitations motivate formulating the problem as a multi-objective optimization problem over two conflicting objectives, the execution time of the workload and the monetary cost of the provisioned Cloud-Edge resources. Because resource performance is uncertain, it is cast in probabilistic terms, with the two objectives treated as random variables rather than fixed values.

In the context of interactive workloads, the allocated resources have to cope with a demand that varies over time and is not known in advance. Autoscaling is a technique that dynamically adjusts the allocated resources by taking two main decisions: when scaling actions are to be triggered, and to what extent resources need to be scaled.

Some autoscaling mechanisms react to changes in monitored performance metrics, such as resource utilization or response time. Other mechanisms instead predict the upcoming workload and adjust the allocated resources accordingly, but rarely align the prediction horizon with the time needed to provision new resources, and few formalize the asymmetry between the two scaling directions. This thesis proposes a predictive autoscaling strategy that matches the prediction horizon to the resource boot time, so that resources provisioned at one control cycle become operational when the demand they target materializes. The strategy is load-aware: it propagates the accumulated backlog of unprocessed requests rather than relying on the arrival rate alone. Scale-out actions are issued promptly to prevent performance degradation, whereas scale-in actions remain conservative to avoid premature deallocation and oscillations.

Despite their differences, the two approaches share a common goal: provisioning the right amount of resources so that performance requirements are met without incurring unnecessary costs.

The main contributions of this thesis can be summarized as follows:

- A probabilistic formulation of a multi-objective optimization problem aimed at identifying the scheduling plans that minimize the execution time of multiple workflows and the monetary cost of Cloud-Edge Continuum resources;
- Design and development of a simulation framework that combines a properly customized Genetic Algorithm (GA) with a Monte Carlo simulator for solving the optimization problem;
- A Load-Aware Predictive Autoscaling Strategy aimed at forecasting the arrival rate and the resources required to ensure the desired utilization level of the resources;
- Design and development of a simulation framework obtained by customizing the CloudSim simulation toolkit for solving the autoscaling problem.
- Extensive simulation experiments based on synthetic and real workloads.

The remainder of this thesis is organized as follows. Chapter 2 introduces the foundational concepts underlying the work, including workload models and the Cloud-Edge Continuum architecture. The state of the art of workflow scheduling and autoscaling is reviewed in Chapter 3.

The resource management problems studied in this thesis and the proposed solution approaches are formulated in Chapter 4.

The workflow scheduling framework is detailed in Chapter 5, while its experimental evaluation on synthetic and real workloads is discussed in Chapter 6.

Chapter 7 describes the Load-Aware Predictive framework, while its experimental results are presented in Chapter 8.

Chapter 9 concludes the thesis and outlines future research directions.

CHAPTER 2

FOUNDATIONAL CONCEPTS

To address the problems of workflow scheduling and autoscaling, formal models of the workload and infrastructure are needed, along with suitable allocation and provisioning methods.

The workloads considered in this thesis take different forms: scientific applications are often modeled as workflows of interdependent tasks, while online services face continuous streams of user requests with fluctuating demand. Workflows are scheduled across infrastructures that may span from Edge devices to Cloud data centers, whereas autoscaling operates within Cloud environments, where resources like Virtual Machines (VMs) or containers can be provisioned and released elastically.

This chapter introduces the models and methods underlying both problems.

2.1 Workload Models

The term workload refers to all inputs (e.g., tasks, requests, queries) submitted to a computing infrastructure. Its processing may be driven by user interaction, with requests arriving continuously and served on demand, or carried out in batch mode, with a finite set of tasks submitted together for processing.

2.1.1 BATCH WORKLOAD

A batch workload consists of scientific applications often represented as workflows consisting of tasks with data and control dependencies.

A standard way to represent a workflow is a Directed Acyclic Graph (DAG). A DAG is defined as $G = (V, E)$, where $V = \{T_1, T_2, \dots, T_k\}$ is the set of nodes representing tasks and $E \subseteq V \times V$ is the set of directed edges representing dependencies among tasks.

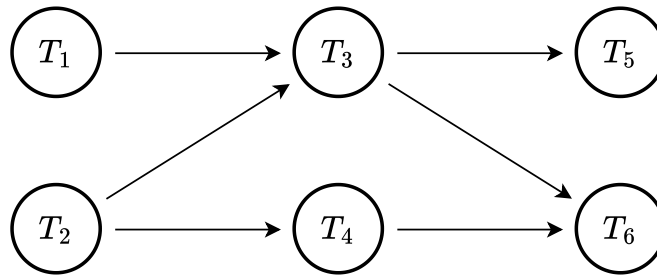


FIGURE 2.1: Example of a workflow represented as a DAG.

Figure 2.1 shows an example of a DAG consisting of six nodes and six edges. For example, the edge from T_2 to T_3 indicates that T_3 cannot start before T_2 completes its execution. Moreover, tasks T_3 and T_4 both depend on the completion of T_2 . The DAG has two entry tasks, T_1 and T_2 , with no predecessors, and two exit tasks, T_5 and T_6 , without successors.

Each task carries a processing demand, and may carry further attributes such as memory and I/O requirements. Each edge $(T_i, T_j) \in E$ is characterized by the volume of data that T_i transfers to T_j . Tasks may also receive input data from external data sources or transfer output data to external devices.

A batch workload often consists of multiple independent workflows, each represented as its own DAG.

2.1.2 INTERACTIVE WORKLOAD

An interactive workload consists of a set of requests sent by users to access services. Each request is characterized by arrival time, processing demand, and additional attributes such as memory or I/O requirements.

The arrival process of the requests is commonly described by their arrival rate or by their interarrival times, defined as the time between two consecutive arrivals.

Requests may reach the system at evenly spaced intervals. However, such regular arrivals are an idealization. Real workloads exhibit irregular patterns, with the arrival rate rising and falling over time in response to, e.g., diurnal usage cycles, user behavior, or external events.

2.2 *Cloud-Edge Continuum*

The Cloud-Edge Continuum is a computing model that organizes infrastructure into layers, spanning resources from the network periphery to centralized Cloud data centers. Devices across the Continuum differ in terms of processing capacity, network performance, and cost.

As Figure 2.2 illustrates, a typical architecture is organized in two computational layers: a Cloud layer and a Fog/Edge layer. The Cloud layer consists of large-scale data centers that offer scalable resources and are interconnected by high-speed backbone links.

The Fog/Edge layer hosts Edge servers, gateways, and smaller regional facilities that offer lower latency to end devices, though with more limited processing capacity.

At the network periphery, IoT devices and end-user equipment interact with the infrastructure but typically do not host sufficient computational resources to execute application tasks. They are therefore not modeled as an infrastructure layer and are considered only as external data sources and storage devices.

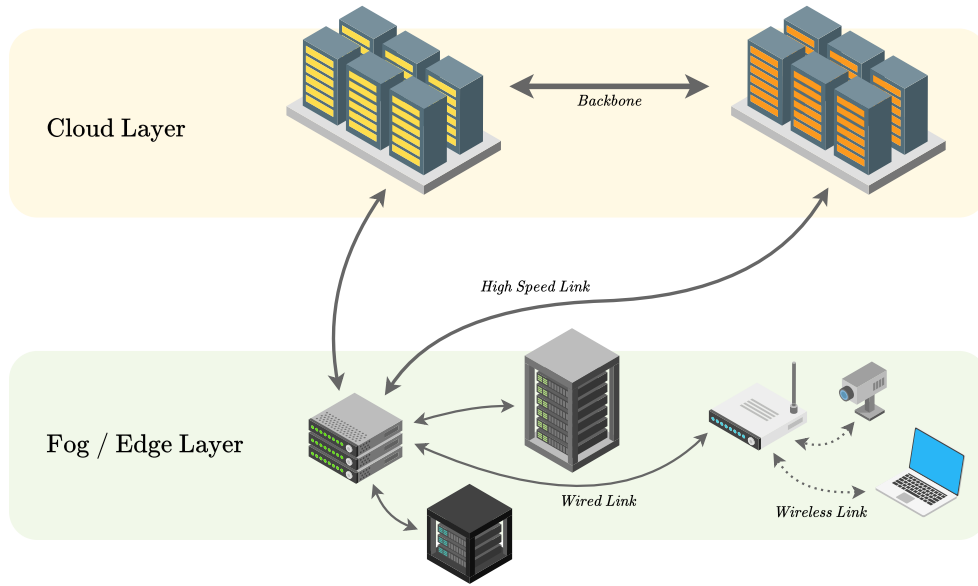


FIGURE 2.2: Layered architecture of the Cloud-Edge Continuum.

According to the NIST definition¹, "*Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources that can be rapidly provisioned and released with minimal management effort or service provider interaction*".

Building on this definition, the Cloud physical infrastructure is shared among multiple tenants, with virtualization technologies providing tenant isolation and a range of service models, e.g., Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS), that abstract away the underlying hardware and allow users to consume resources at different levels of granularity.

Cloud providers adopt pay-as-you-go pricing models that meter the type, quantity, and duration of consumed compute, storage, and network resources.

The Fog/Edge layer spans resources of different scales along the path from Cloud data centers to end devices.

The term *Fog computing* was originally introduced by Cisco [2] to describe an intermediate layer between the Cloud and end devices. Although *Fog computing* and *Edge computing* are frequently used interchangeably, they denote distinct segments of this layer: Fog generally refers to infrastructure deployed at the level of regional or metropolitan networks, whereas Edge refers to re-

¹P. Mell and T. Grance, *The NIST Definition of Cloud Computing*, NIST Special Publication 800-145, 2011. <https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-145.pdf>

sources located at the boundary of access networks or directly within local sites.

These resources are typically operated by network providers, enterprises, or vertical service operators, and may be dedicated to a single organization or offered under pricing schemes that differ from traditional Cloud models.

2.3 Workflow Scheduling

Workflow scheduling is the process of mapping tasks to resources and determining the task scheduling order with the goal of optimizing one or more performance objectives while satisfying task dependencies.

In Cloud-Edge environments, where resources are heterogeneous and distributed, workflow scheduling is formulated as an optimization problem. Its solutions, i.e., scheduling plans, specify the mappings of tasks to resources and feasible execution orders that satisfy task dependencies.

Objectives to be optimized include makespan, defined as the total elapsed time from the start of the first task to the completion of the last, monetary cost for leasing computation and communication resources, energy consumption and reliability.

These objectives are often in conflict. Reducing makespan typically requires mapping tasks to faster or more numerous resources, thus increasing cost. Conversely, minimizing cost entails using slower or fewer resources, which leads to a longer makespan.

2.3.1 PROBLEM CLASSIFICATION

Workflow scheduling problems can be classified along several criteria that influence the formulation and the choice of the solution approach.

Most classical scheduling formulations assume deterministic task execution times and communication delays, under which the objective values of a scheduling plan can be obtained exactly. In contrast, stochastic formulations take into account performance variability, that is, fluctuations in the actual processing capacity of compute resources and in the latency and throughput of network links.

In single-objective formulations, the scheduling problem is aimed at optimizing a single metric. Any two candidate scheduling plans are comparable by their objective value, and the notion of an optimum is unambiguous.

When multiple conflicting objectives are considered simultaneously, the goal of the scheduling problem shifts from finding an optimal solution to identifying a set of solutions that represent the best achievable tradeoffs among objectives.

The distinction between offline and online scheduling refers to when scheduling decisions are made. In offline scheduling, full information about the workload is assumed to be available ahead of its execution. This typically means that the complete DAG structure, task demands and data transfer sizes are known.

Offline methods are particularly suited to batch scientific workloads where the goal is to find the "best" trade-offs before their deployment.

By contrast, in online scheduling, decisions are taken incrementally during execution. Resources and running tasks are continuously observed and task-to-resource mappings are updated as new information becomes available, such as actual task completion times or changes in resource availability. Online scheduling can respond to deviations from expected conditions, such as resource slowdowns or failures. However, this approach operates under time pressure and incomplete information, often relying on fast heuristics, thus making it harder to achieve optimal scheduling plans.

2.3.2 SOLUTION APPROACHES

Workflow scheduling is known to be NP-hard [3] even in its most basic formulations. Exact methods are therefore computationally feasible only when the number of tasks and resources is small. Hence, in real scenarios, heuristics or meta-heuristic approaches are adopted instead.

HEURISTIC APPROACHES

List-based heuristics like the Heterogeneous Earliest Finish Time (HEFT) algorithm [4] rely on task prioritization and resource selection strategies to build efficient scheduling plans. In particular, HEFT consists of two main phases.

In the first phase, a priority value, known as the "upward rank", is assigned to each task of the DAG. The value is an estimate of the longest path length from a task to the exit node and is computed recursively by traversing the DAG

backwards from its exit nodes. For task v , the rank is defined as:

$$Rank(v) = \bar{w}_v + \max_{u \in succ(v)} (c_{v,u} + Rank(u))$$

where $\bar{w}_v$ is the average execution time of v across all available resources and $c_{v,u}$ is the communication time between v and successor u . Tasks closer to the critical path of the workflow, i.e., the longest path from the entry to the exit node in terms of cumulative execution and communication time, are assigned higher priorities. Once all ranks are computed, tasks are ordered in decreasing priority.

In the second phase, tasks are scheduled one-by-one according to their priority. When scheduling a task, HEFT computes its Earliest Finish Time (EFT) on every available resource and assigns the task to the resource yielding the smallest value, thereby greedily reducing the contribution of each task to the overall makespan.

Because HEFT makes decisions greedily, it runs in polynomial time. Multi-Objective Heterogeneous Earliest Finish Time (MOHEFT) [5] variants extend HEFT to multi-objective contexts.

META-HEURISTIC APPROACHES

While heuristics construct a single scheduling plan via a greedy procedure, meta-heuristics iteratively maintain and evolve a population of candidate solutions according to a fitness function. Exploration of the solution space is guided by operators that introduce variation, enabling broader coverage of large combinatorial spaces at the expense of higher computational cost.

The scheduling plan of a workflow DAG is first encoded into a data structure (e.g., a vector). From this encoding, a fitness evaluation procedure reconstructs a scheduling plan and computes its objective values. Search operators specific to the meta-heuristic are then applied iteratively to improve the population, until a termination criterion is met and the best solutions are returned.

Genetic Algorithms (GAs) [6] mimic natural selection by evolving a population of candidate solutions (chromosomes) over successive generations through selection, crossover, and mutation operators. Selection favors individuals with better fitness, crossover recombines portions of two parent chromosomes to produce offspring, and mutation introduces small random perturbations to

maintain diversity. In multi-objective settings, GAs are combined with selection mechanisms that rank individuals by objective values jointly and by spread across the objective space, ensuring that the final population covers a range of tradeoffs.

Another approach is the Particle Swarm Optimization (PSO) [7] that treats each candidate scheduling plan as a particle moving in a continuous search space. At each iteration, particles update their positions using inertia, cognitive attraction toward their own best-known position, and social attraction toward the best-known position within the swarm.

Multi-objective variants of PSO generalize this mechanism to handle multiple objectives jointly [8].

Ant Colony Optimization (ACO) [9] frames solution construction as a path-building process. Artificial ants incrementally build scheduling plans by probabilistically selecting the next task to schedule and the resource for its execution, with preferences biased by pheromone values (reflecting past solution quality) and heuristic information (reflecting estimated impact on objectives). After each construction round, pheromone is reinforced on choices that appear in the best solutions and evaporated globally, progressively biasing future ants toward promising regions of the search space while preventing premature convergence.

Other meta-heuristics that can be applied to workflow scheduling include Firefly Algorithms [10], Artificial Bee Colony Optimization [11], and Salp Swarm Optimization [12].

Despite their differences, these methods share the same conceptual ingredients: a DAG-based encoding of scheduling plans, a fitness evaluation procedure that may be single- or multi-objective, and domain-specific operators or heuristics to maintain feasibility and exploit problem structure. In addition, many meta-heuristic approaches are integrated with list-based heuristics as initialization or local search components to enhance convergence.

2.4 Resource Provisioning with Autoscaling

As introduced in Section 2.2, Cloud environments provide elastic capacity. While this allows resource allocation to be adjusted to varying workloads and

performance requirements, the amount of capacity to be provisioned must be carefully managed.

Static provisioning, where a fixed amount of resources is reserved in advance, is rarely adequate to cope with time-varying interactive workloads. The choice of the "right" amount of resources to be allocated is critical, as it can either lead to over-provisioning or under-provisioning conditions.

Over-provisioning occurs when the capacity of the resources exceeds what is actually required. In this situation, service-level objectives are always satisfied, but the utilization of the resources is low. Over-provisioning therefore leads to unnecessary monetary costs.

In contrast, under-provisioning conditions occur when the workload demand exceeds the capacity of the allocated resources. While under-provisioning tends to minimize costs, it does so at the expense of the performance and can lead to reputational damage or penalty costs.

To overcome over- and under-provisioning, autoscaling mechanisms serve to dynamically provision resources. The main goal of autoscaling is to continuously regulate the allocated resources as to minimize over-provisioning and under-provisioning by keeping the capacity close to the minimum required to satisfy the predefined performance requirements.

2.4.1 CONCEPTUAL MODEL

Autoscaling is an automatic closed-loop control mechanism that exploits Cloud elasticity by dynamically adjusting the number of resources in response to workload changes.

Commonly used conceptual models for such loops are MAPE [13] and MAPE-K [14], which identify four primary phases, namely Monitor, Analyze, Plan, Execute, optionally operating over a shared Knowledge base. Figure 2.3 illustrates the interactions between the main phases of the MAPE-K loop.

The Monitor phase is responsible for collecting raw data such as application-level metrics (e.g., arrival rate, processing rate, and response time) as well as resource-level metrics (e.g., CPU, memory, and network utilization).

The Analyze phase processes the monitored data to derive information useful for making decisions. This may involve simple statistical approaches or more sophisticated approaches based on machine learning.

The Plan phase determines whether and which scaling action should be

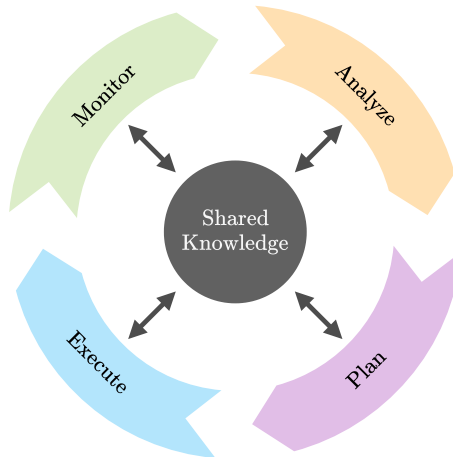


FIGURE 2.3: Representation of a MAPE-K control loop.

taken, answering two questions: *when* to scale, that is, the timing of scaling actions, which can be immediate or delayed to avoid oscillations; and *to what extent* to scale (e.g., how many resources to add or remove). Planning can rely on rule-based policies, optimization models, queuing-theoretic formulas, or learned policies such as reinforcement learning agents.

The Execute phase carries out the planned scaling actions by interacting with the underlying infrastructure. This phase must account for actuation delays, that is, resource boot or shutdown times.

The shared Knowledge component, which distinguishes MAPE-K from the simpler MAPE model, is a persistent information base accessible to all four phases. It typically holds historical monitoring data, configuration parameters refined over time, and learned models, so that each phase can draw on past observations and shared state rather than operate on the latest measurement alone.

2.4.2 SCALING ACTIONS

Two types of scaling actions can be identified:

- *Scale-out* actions denote an increase in the available capacity, obtained by provisioning additional resources;
- *Scale-in* actions denote a reduction in the available capacity, typically by releasing existing resources, often after draining their current workload to avoid interrupting in-flight requests.

The most common form of scaling is horizontal, in which the number of allocated resources is increased or decreased. Horizontal scaling is often supported by Cloud providers and is often preferred when resources can be started and stopped relatively quickly.

An alternative is vertical scaling, which changes the capacity of individual resources (e.g., CPU, memory) rather than their number.

2.4.3 CONTROL STRATEGIES

A common taxonomy distinguishes autoscaling policies in three categories, namely: reactive, proactive and hybrid approaches.

Decisions of reactive autoscaling policies are based on the current or recent past state of the monitored resources. A canonical form is threshold-based control, in which scaling actions are triggered when monitored metrics cross predefined bounds. For example, a reactive policy might trigger a scale-out action if CPU utilization exceeds a specified upper threshold for a certain duration. On the other hand, if utilization falls below a lower threshold, the autoscaler triggers a scale-in action.

Reactive schemes may use resource-level metrics (CPU, memory, network) or application-level metrics such as response time. The main advantages of reactive approaches are their simplicity, low computational overhead, and easy deployment.

Proactive autoscaling extends the control horizon by incorporating predictions of future workloads or resource usage into the decision process. The central idea is to allocate additional capacity before being requested, thereby mitigating or avoiding performance degradation due to provisioning delays.

The prediction component can be built in several ways, for example via time series methods (e.g., ARIMA, exponential smoothing, seasonal decomposition) or machine learning (e.g., regression models, random forests, neural networks). Predictive methods can significantly improve autoscaling performance under workloads with predictable patterns, such as daily or weekly patterns. However, their effectiveness depends on forecast accuracy.

Hybrid autoscaling combines reactive and proactive approaches to exploit the strengths of both. Reactive mechanisms offer immediate responses to observed changes, while proactive components forecast capacity adjustments based on trends and recurring patterns.

CHAPTER 3

STATE OF THE ART

This chapter surveys the body of literature most closely related to the contributions presented in this thesis. The discussion is organized around the two main themes that, together, underpin the efficient execution of computational workflows in modern distributed infrastructures, namely, workflow scheduling and autoscaling.

In particular, Section 3.1 addresses the workflow scheduling problem, reviewing how researchers have tackled the challenge of mapping interdependent tasks onto heterogeneous and distributed resources. Starting from single workflow scheduling in Cloud environments, the discussion progressively broadens its scope to encompass hybrid and Cloud-Edge infrastructures, the concurrent scheduling of multiple workflows, and approaches that explicitly account for performance uncertainty.

Section 3.2 shifts the focus to autoscaling techniques in Cloud environments, examining how computational resources can be dynamically provisioned and released to match fluctuating workload demands. The review is organized according to the timing and nature of scaling decisions, covering reactive, proactive, and hybrid approaches.

Although scheduling and autoscaling are often studied in isolation, they are in practice deeply intertwined: scheduling decisions determine how tasks are placed on resources, while autoscaling decisions govern which and how many resources are available in the first place. A holistic understanding of the contributions in both areas is therefore essential to motivate and contextualize the contributions of this thesis.

3.1 Workflow Scheduling in the Cloud-Edge Continuum

Over the years, the problem of workflow scheduling has been extensively addressed in the literature, gaining increasing attention thanks to the proliferation of Cloud computing and, more recently, Edge and IoT computing paradigms.

As already stated (see Section 2.1), workflows are generally modeled as DAGs, where nodes represent tasks and edges denote dependencies among them. The primary goal of workflow scheduling is to determine an optimal mapping of tasks onto available resources with the aim of optimizing various objectives, such as execution time, monetary cost, energy consumption, or reliability. This has to be done while satisfying constraints like budget and deadlines.

In solving this problem, researchers face several challenges, including task dependencies, resource heterogeneity, limited resource availability, and conflicting objectives. Additionally, the dynamic nature of Cloud and Edge environments introduces uncertainty in resource performance. This further complicates the scheduling process. Moreover, as the size of workflows increases, the search space of potential mappings grows exponentially, making exact solutions computationally infeasible. Consequently, a variety of heuristic and meta-heuristic algorithms have been proposed in the literature for both Cloud and Edge environments [15].

In what follows, the existing literature is explored by focusing on single workflow scheduling in Cloud environments, reviewing studies that approach the problem from different angles with different optimization objectives and constraints. The scheduling problem is then examined in more complex infrastructures, namely hybrid Clouds and the Cloud-Edge Continuum. Finally, studies addressing the concurrent scheduling of multiple workflows, as well as those accounting for performance uncertainty in the scheduling process, are reviewed.

3.1.1 WORKFLOW SCHEDULING IN CLOUD ENVIRONMENTS

In this section, the literature on workflow scheduling in Cloud environments, a topic extensively reviewed in existing surveys [16–20], is analyzed by focusing on the problem formulations.

Many works formulate workflow scheduling as a single-objective optimization problem aimed at minimizing execution cost of the workflow while satisfying a deadline (e.g., [21–26]). Other works (e.g., [27–31]) formulate workflow scheduling as a multi-objective optimization problem balancing trade-offs between execution time and cost. Finally, other studies consider optimization objectives beyond time and cost (see, e.g., [32–34]), such as energy consumption, reliability, and resource utilization, alongside traditional objectives.

Independently of the specific optimization problem addressed, it can be observed that most works leverage heuristic and meta-heuristic algorithms to find near-optimal scheduling plans in a short time. As already mentioned in Section 2.3, commonly used techniques include list-based heuristics like HEFT as well as meta-heuristics such as GA, PSO, and ACO. To effectively handle task dependencies and resource heterogeneity, these algorithms are often customized with problem-specific encoding schemes, fitness evaluation schemes, and initialization strategies.

For example, Rodriguez and Buyya [21] formulate the workflow scheduling problem as a single-objective optimization problem aimed at minimizing the monetary cost of executing a scientific workflow while satisfying a user-defined deadline. A DAG is used to model the workflow structure, and the Cloud model consists of a set of heterogeneous VMs of different types and with different capabilities. Each VM type is characterized by its computational capacity, cost per time unit, and boot time overhead.

In this setting, the workflow scheduling problem is solved by taking advantage of the PSO meta-heuristic. Scheduling plans encode task-to-resource mappings and are represented by particles in an N -dimensional search space where N is the number of tasks in the workflow. The coordinates of each particle identify the resource assigned to each task. Particles are evaluated by computing task execution times according to the characteristics of the assigned resources. Task start and finish times are then derived from the execution time and the completion time of predecessor tasks. Cost is directly encoded in the fitness function as the objective to minimize and is computed as the sum of each task's execution time multiplied by the cost per time unit of its assigned resource. Makespan corresponds to the finish time of the last task to complete in the workflow and is used to verify satisfaction of the deadline constraint.

To compare solutions and satisfy the deadline constraint, the well-known non-dominated sorting strategy from Deb et al. [35] is applied. Solutions satisfying the deadline are considered feasible and are always preferred over infeasible ones despite their cost, while among infeasible solutions, those with

lower levels of constraint violation (i.e., makespan exceeding the deadline by a smaller margin) are preferred.

In order to satisfy strict deadline constraints, Verma and Kaushal [22] approach the deadline-constrained cost-minimization workflow scheduling problem by leveraging the structure of workflow DAGs. In this work, the execution order of tasks is explicitly modeled in the scheduling plans along with the task-to-resource mappings. A 2D representation is proposed, where each row corresponds to a resource and each column represents the tasks assigned to that resource according to the execution order.

To this end, an initial set of schedules is generated by ordering the tasks according to their bottom- and top-level ranking within the workflow. The bottom-level of a task is defined as the longest possible time to complete the workflow starting from that task. Conversely, the top-level of a task is the longest possible time to reach it from an entry task. These definitions are used to identify tasks on the critical path of the workflow, which are scheduled first to satisfy the deadline.

A GA is employed to search for scheduling plans. In this context, the initial set of schedules is used to improve the quality and diversity of the GA population. To evaluate a scheduling plan, the proposed fitness function is defined as the ratio between the makespan of the scheduling plan and the user-defined deadline, thereby capturing both objective minimization and deadline satisfaction in a single value. The deadline is then enforced by retaining only schedules whose fitness value is less than one.

Other papers aim to improve GA-based workflow scheduling by guiding the search process during evolution to favor either exploration, meaning the discovery of new and diverse regions of the search space, or exploitation, meaning the refinement of promising solutions already identified. For instance, Liu et al. [24] apply this concept to a workflow scheduling problem aimed at minimizing execution cost under a deadline constraint. This work introduces an adaptive penalty factor in the fitness function to handle the deadline constraint. The authors observe that smaller penalty factors in the initial evolution stages favor exploration of the search space by allowing infeasible solutions to survive longer in the population. As the evolution progresses, the penalty factor is gradually increased to enforce the deadline constraint. In the Genetic Algorithm, the probability of applying crossover and mutation operators is also modified according to the population diversity. If many individuals with poor fitness are present, mutation probability is increased while crossover probability is decreased. Conversely, if the population contains many individuals with

good fitness values, both probabilities are decreased.

Selection of operator probabilities is left to a “controller” population that co-evolves alongside the main population of scheduling plans. Each individual in the controller population encodes a pair of crossover and mutation probabilities that are mapped to a set of scheduling plans in the main population. Selection of the best operator probabilities is based on the best fitness and proportion of feasible individuals among the corresponding scheduling plan set.

In the context of deadline-constrained cost-minimization problems whose solution is based on GAs, Iranmanesh and Naji [26] suggest a hybrid scheduler combining GA with HEFT to obtain fast convergence speed, while also employing load-balancing to better distribute the workload across available resources. The structure of the workflow DAG is exploited to identify independent workflow partitions where tasks can be processed concurrently. Similarly to [22], this work leverages top- and bottom-level rankings to stratify the workflow DAG into layers, thus identifying partitions of independent tasks. This partitioning scheme is used to guide both the initial population generation and the design of custom operators for the GA.

Regarding the initial population, scheduling plans are generated by prioritizing tasks in the critical path of each partition, which are scheduled first on high-performance resources. The initial population is also integrated by a single scheduling plan generated by the HEFT heuristic to provide a strong baseline solution to the GA.

In the context of genetic operators, a new “inversion” operator is introduced to shuffle the order of tasks within each partition and explore different scheduling plans. Multiple variants of crossover and mutation operators are applied in parallel, and only the offspring that provides the greatest fitness improvement over the parents is retained. Specifically, single-, two-, and three-point crossover operators are evaluated, and the offspring with the highest resulting fitness is selected. Likewise, for mutation, single- and two-point mutations are applied, and the mutation producing the best fitness improvement is preserved.

Some recent works have formulated the workflow scheduling problem as a multi-objective optimization problem aimed at simultaneously minimizing makespan and monetary cost. This formulation allows practitioners to explore the inherent trade-offs between time and cost, as reducing execution time often leads to increased costs due to the use of a larger number of high-performance resources, and vice versa.

In this context, workflow scheduling is typically approached as a Pareto optimization problem, where the concept of dominance is used to compare schedul-

ing plans by simultaneously considering both objectives. A scheduling plan dominates another if it is not worse in any objective and strictly better in at least one. Consequently, Pareto-optimal solutions are those that are not dominated by any other found solution. These solutions are collected in a Pareto front representing the best trade-offs between time and cost.

A Pareto-based approach is applied by Zhu et al. [27] in the formulation of a multi-objective workflow scheduling problem. The Pareto front of non-dominated scheduling plans is updated throughout the evolution process of a GA, and provides high-quality individuals that guide the search towards better trade-offs. The authors propose a problem-specific encoding scheme where each scheduling plan is composed of three components. First, a task sequence defines the execution order of tasks while respecting their precedence constraints. Second, a mapping component assigns each task to a specific VM. Third, a VM type selection component specifies the type of VM to be used for each task.

To ensure that task dependencies are respected, the proposed framework explicitly leverages a topological sort of the workflow DAG. This allows the GA to generate and validate task sequences that satisfy dependencies. Building on this representation, custom crossover and mutation operators are designed to effectively deal with task dependencies while maintaining a valid topological order in the offspring scheduling plans. This aspect is critical in workflow scheduling, as precedence constraints must always be satisfied to guarantee feasible execution plans.

Aziza and Krichen [28] introduce deadline and budget constraints alongside makespan and cost minimization. The authors propose a hybrid GA where the initial population is seeded by HEFT to accelerate convergence toward better cost–time trade-offs. In this work, the fitness function combines both objectives into a single weighted sum of time and cost components, each being the ratio of the objective over its constraint. While this approach simplifies the optimization process by reducing it to a single-objective problem, it requires careful tuning of the weights to balance the two objectives.

A different approach for simultaneously minimizing the cost and makespan is proposed by Chen et al. [30], who apply a multi-objective ACO algorithm employing two separate populations, one for optimizing execution time and the other for cost. The populations share a global archive of non-dominated solutions to guide the search. A pheromone update rule is designed to balance the search of both objectives by considering the set of non-dominated solutions

in the archive. Additionally, a complementary heuristic strategy is employed to avoid each colony focusing solely on its single objective. The authors also introduce an “Elite Study” strategy to improve the quality of solutions in the global archive by applying small- and large-scope mutations to elite individuals, enhancing exploration around high-quality solutions.

Belgacem and Beghdad-Bey [31] propose a hybrid algorithm that combines the HEFT heuristic for precedence-aware task ordering with ACO for resource allocation to jointly minimize makespan and monetary cost. While HEFT lays the initial task order, the pheromone update rule implements Pareto ranking and crowding distance to maintain a well-spread Pareto front and obtain a wide range of trade-off solutions.

Makespan and cost are not the only objectives of interest in Cloud workflow scheduling. Wang et al. [32] introduce reliability as a first-class objective alongside makespan. This work aims at minimizing the makespan of a workflow and maximizing its reliability by employing a reliability-driven reputation model to evaluate resource failures and ensure that tasks are assigned to reliable resources. For this purpose, the authors associate with each Cloud resource an hourly failure rate. The reliability objective is computed as the total execution time of tasks weighted by the failure rates of the resources assigned to each task.

The problem solution is sought through a custom GA with a look-ahead strategy, which evaluates task execution order on each resource at fitness evaluation time. In fact, in the proposed approach, scheduling plans only encode task-to-resource mappings, while the execution order of tasks is determined by a heuristic applied before computing makespan and reliability. This heuristic enqueues tasks to resources according to their computational requirements and the longest path in the workflow DAG. This approach effectively prioritizes tasks with longer paths to completion, ensuring that tasks with a larger impact on workflow makespan are scheduled first.

Verma and Kaushal [33] introduce energy consumption as a third objective to be minimized alongside makespan and cost, while satisfying deadline and budget constraints. In particular, energy consumption is computed by considering the power consumption of each VM type over time and its utilization (i.e., being idle or busy executing tasks) according to the Dynamic Voltage and Frequency Scaling (DVFS) power management technique.

To solve the scheduling problem, the authors hybridize a multi-objective PSO with a heuristic based on HEFT. In particular, the heuristic is applied to gener-

ate a part of the initial population, while PSO refines the solutions through its search process. The multi-objective PSO employs an external archive of non-dominated solutions updated via dominance sorting. A parameterized density estimator is used to maintain diversity among the obtained solutions while the deadlines are enforced at each iteration, meaning that only scheduling plans meeting constraints are added to the archive. To conclude, a mutation operator randomly modifies scheduling plans with a probability decreasing over iterations to improve exploration and avoid premature convergence.

Xia et al. [29] also formulate workflow scheduling as a multi-objective problem aimed at minimizing makespan and energy consumption. In particular, energy consumption is computed at the VM level, by considering both active and idle power consumption over time. The search for solutions is carried out by a meta-heuristic based on Non-dominated Sorting Genetic Algorithm II (NSGA-II)[35], where scheduling plans are encoded in chromosomes composed of two parts: task execution order and task-to-VM mappings.

To provide good starting points for the search, the initialization scheme considers the amount of data transferred by tasks, assigning tasks with higher data demands to higher capacity VMs. Custom genetic operators are designed to act on both the provisioning and order parts of the chromosome while preserving task dependencies in the execution order.

The top- k individuals, that is, the k scheduling plans with the best fitness values, are analyzed after each NSGA-II iteration to identify common subsequences in their task execution order. These subsequences represent favorable task orders, and chromosomes containing them receive lower crossover and mutation probabilities. This technique helps preserve high-quality portions of the scheduling plans, balancing exploration and exploitation during the evolutionary process.

Zeedan et al. [34] formulate an optimization problem with three objectives, aimed at minimizing workflow makespan and monetary cost, and maximizing resource utilization to ensure that the allocated resources are effectively used during workflow execution. Utilization is computed as the average task execution time for all tasks and resources over workflow makespan.

The authors solve the optimization problem by proposing a hybrid algorithm that combines three components: a greedy randomized adaptive search procedure, a binary artificial bee colony algorithm, and the HEFT heuristic. The greedy procedure is used to build an initial population of scheduling plans by iteratively assigning tasks to VMs based on their expected finish time or cost. The binary artificial bee colony algorithm refines the initial population by ex-

ploring the solution space through local search and neighborhood exploration. HEFT is employed to determine the task scheduling order according to the allocations identified by the bee colony.

3.1.2 WORKFLOW SCHEDULING IN THE CLOUD-EDGE CONTINUUM

As already mentioned in Section 2.2, the widespread adoption of the Cloud paradigm has introduced flexibility and scalability in computing applications, including scientific workloads. However, the use of Cloud infrastructures may not be suitable for all types of applications and scenarios. Additionally, the pay-as-you-go model of public Clouds may lead to unpredictable and high operational expenses for data-intensive applications with significant data transfer requirements. The Cloud-Edge Continuum paradigm aims at bridging the gap between centralized Cloud data centers and decentralized Edge and IoT devices, envisioning a more integrated and collaborative environment where diverse computing resources enable seamless data processing and resource management.

In this section, the literature on workflow scheduling beyond public Clouds is reviewed, first considering workflow scheduling in private Cloud [36], in hybrid Cloud-Fog [37–39], and in Edge computing scenarios [40–42].

For instance, Zhou et al. [36] approach the workflow scheduling problem in an infrastructure consisting of both private and public Cloud resources. Private Cloud resources are assumed to be free of charge, although limited both in terms of performance and quantity. On the other hand, public Clouds are more numerous and powerful, but incur monetary costs based on their usage. In addition, data transfers between private and public Clouds also contribute to the overall monetary cost of workflow execution.

The main contribution of this work consists in the formulation of two workflow scheduling problems: the first is a single-objective problem that minimizes execution cost under a user-defined deadline, and the second is a multi-objective problem that jointly minimizes execution cost and makespan. To solve these problems, the authors design two GA-based heuristic algorithms with tailored genetic operators. In the first heuristic, both the cost objective and the deadline constraint are integrated into the fitness function, computed as the execution cost inflated by a penalty factor when the deadline constraint is violated. The second heuristic extends the previous one to a multi-objective optimization problem that simultaneously minimizes execution time and cost

without constraints. In this case, the fitness evaluation considers both objectives, using Pareto dominance to compare the obtained scheduling plans.

Fog computing has emerged as a complementary paradigm to Cloud computing, bringing computation and storage resources closer to the data sources and end-users. However, the resource heterogeneity and distribution that the Fog introduces pose significant challenges to workflow scheduling, as surveyed from a modeling perspective in [43] and from an algorithmic perspective in [44].

Many works have addressed the problem of scheduling in Fog environments considering various optimization objectives such as network latency [45–48], resource utilization [49], energy consumption [50], makespan and monetary cost [51–54]. As observed in the taxonomy [55], these works primarily focus on independent tasks rather than complex workflows with task dependencies.

Being able to schedule complex workflows in a Cloud-Fog environment is essential to fully leverage the potential of this paradigm for real-time and latency-sensitive applications, as highlighted in the systematic review [56].

Pham et al. [37] consider an infrastructure based on the collaboration of Cloud and Fog resources. The authors discuss how it is essential to strategically distribute the workload among the Cloud and the Fog to get the most out of both. The workflow scheduling problem is formulated as an optimization problem minimizing makespan and monetary cost with a deadline constraint. The proposed heuristic refines scheduling plans by employing a strategy that reassigns tasks according to the critical path of the workflow DAG. To reduce the overall makespan of the workflow, tasks on the critical path are scheduled for execution on the most performant resources, while to minimize monetary cost, non-critical tasks are reassigned to lower-cost resources.

A three-objective scheduling problem aimed at minimizing workflow makespan, execution cost, and energy consumption in Cloud-Fog computing is formulated by Singh and Chaturvedi [39]. Energy consumption is computed according to two separate models for active and idle resource states. While the former accounts for the per-resource energy consumption during task execution, the latter considers the minimum energy consumption when resources are idle. In both cases, energy consumption is computed as a product of voltage supply level and operating frequency according to resource specifications.

Execution cost encompasses both computational and communication costs, the latter being particularly relevant in Cloud-Edge environments due to the pricing models typically adopted by Cloud providers for data transfers in and

out of their data centers.

The optimization problem is solved using a hybrid GA-PSO method since GA-based meta-heuristics tend to outperform other algorithms after a significant number of iterations, whereas PSO-based methods typically converge faster in the early stages of the search. In the initial phase, the GA explores the search space and produces a population of candidate solutions. Subsequently, the PSO is applied to refine the best solutions generated by the GA.

The Edge computing paradigm further extends computational resources closer to end-users and data sources, often leveraging the capabilities of IoT devices themselves. Xie et al. [40] address the workflow scheduling problem in the Cloud-Edge Continuum by designing a framework able to schedule tasks across heterogeneous resources while optimizing both makespan and cost. The considered infrastructure is composed of three layers: Cloud, Edge, and terminal layers. The latter mainly consists of smart IoT devices such as smartphones, wearables, and smart home appliances. These devices send service requests to the Edge layer, which is composed of routers, switches, and gateways, and is employed to process latency-sensitive tasks. The Edge layer, in turn, offloads latency-tolerant and computation-intensive tasks to the Cloud layer.

The optimization problem is solved using a variant of the PSO algorithm, modified to improve its exploration capabilities in the Cloud-Edge Continuum environment. In particular, the authors introduce non-linearity in the particle inertia computation and modifications to the search, selection, and mutation processes. This results in better performance in minimizing the makespan and cost of scheduling plans compared to other PSO variants.

A similar infrastructure is considered in Ijaz et al. [38], where the focus is on scheduling workflows by simultaneously minimizing both makespan and energy consumption under a deadline constraint. This work discusses the importance of energy efficiency in Cloud-Edge environments, especially in the case where Edge and IoT devices are battery-powered and deployed in areas with limited and unreliable energy sources. Energy consumption is modeled according to DVFS, exploiting idle slots between task executions to reduce energy consumption.

The optimization problem is solved using a heuristic working in two phases. First, tasks are mapped across Cloud and Edge resources, favoring Edge resources for lightweight and time-sensitive tasks and Cloud resources for compute-intensive tasks. Secondly, a deadline-aware frequency scaling procedure adapts the processing capacity of each node to reduce energy consumption while satisfying task deadlines.

Bacanin et al. [41] use a similar infrastructural model as a basis to schedule workflows with the objective of minimizing overall makespan and monetary cost. In particular, task communication delays are explicitly modeled and integrated in the computation of both objectives. This is particularly important in Cloud-Edge environments, where Cloud data centers are usually connected through high-speed backbone networks, while communication between Edge nodes and Cloud data centers often relies on public Internet connections that may introduce significant latency and bandwidth limitations.

The optimization problem is solved by a Firefly Algorithm tailored to workflow scheduling. Candidate scheduling plans encode task-to-resource mappings, while the fitness calculation is communication- and cost-aware.

Solution search is enhanced by including genetic operators to increase the exploration and exploitation abilities of the algorithm. In addition, quasi-reflection learning is introduced to improve population diversity. This technique consists of generating new scheduling plans by assigning resources with opposite capabilities to the resources assigned in existing solutions. This allows the algorithm to explore under-sampled regions of the search space.

Alsadie and Alsulami [42] also employ a modified Firefly Algorithm to identify scheduling plans with minimal makespan and monetary cost. This is achieved thanks to placement decisions that jointly consider computation time, monetary cost, as well as communication delays between Cloud and Edge resources. The problem formulation integrates heterogeneity and data-locality concepts to capture the characteristics of Cloud-Edge environments.

The Firefly Algorithm is tailored to DAG scheduling by using a precedence-aware encoding that maps tasks to resources, a composite fitness function that couples execution and communication times, and algorithmic refinements such as adaptive movement parameters and a lightweight local-improvement step to maintain feasibility and escape local optima.

3.1.3 MULTIPLE WORKFLOW SCHEDULING

While the majority of the existing literature extensively covers the scheduling of single workflows, in real-world scenarios multiple workflows need to be scheduled concurrently on shared resources. For example, in scientific computing environments, multiple workflows representing different experiments or analyses that need to be executed by a shared computational infrastructure are often submitted.

The scheduling of multiple workflows brings challenges not present when dealing with single workflows. In particular, resource contention becomes a significant concern. Tasks from different workflows compete for the same set of resources, leading to potential delays and increased makespan. Therefore, scheduling algorithms have to devise strategies to mitigate this contention. This has led to the emergence of a new branch of workflow scheduling research that specifically addresses multi-workflow scenarios [57, 58], as reviewed in the survey [59].

One of the pioneering studies is Zhao and Sakellariou [60], where multiple workflows are scheduled concurrently on a shared set of resources with the objective of maximizing schedule fairness. The concept of fairness is introduced to ensure that no single workflow is overly disadvantaged due to resource sharing.

Fairness is measured according to the slowdown of each workflow, that is, the ratio between the execution time of a workflow when running concurrently with other workflows and its execution time when running in isolation. To solve the optimization problem, the authors propose a heuristic that schedules first tasks from workflows with the highest degree of slowdown, thereby balancing the slowdowns across all workflows. Two policies are introduced to compute the slowdown. In the first policy, slowdown for a workflow is recomputed after each scheduling decision involving its tasks, while in the second, slowdown for all workflows is recomputed after any scheduling decision within the system.

Bittencourt and Madeira [61] go one step further by studying the trade-off between overall performance (i.e., makespan) and fairness in multi-workflow scheduling.

Four different scheduling strategies are proposed. The first one consists of a sequential procedure that schedules workflows one after the other, optimizing only the makespan of individual workflows without considering fairness. The second one attempts to fill idle slots in the schedule of already scheduled workflows with tasks from other workflows, thus improving resource utilization and overall makespan. The third strategy schedules tasks from different workflows in a round-robin fashion, aiming at balancing the execution of all workflows and improving fairness. The fourth strategy merges all the workflows in a single DAG by adding dummy entry and exit tasks to handle multiple entry and exit points.

Strategies are applied within the proposed path scheduling heuristic, in charge of task ordering and resource assignment. The heuristic groups tasks

into clusters with the aim of scheduling them on the same resource. This reduces communication overhead, since tasks on the same resource can share data through local storage instead of transferring it over the network. To define clusters, the DAG topology is analyzed and tasks that can be scheduled together are identified.

Bochenina et al. [62] extend [61] by further exploiting idle gaps in scheduling multiple workflows. The authors observe that idle periods may arise when tasks cannot be scheduled immediately due to dependencies that need to be resolved first, leading to underutilized resources.

To address this issue, the proposed multi-workflow scheduling strategy opportunistically fills idle periods in a workflow schedule by borrowing tasks from other workflows in the system, thereby improving overall resource utilization and reducing idle times.

Three heuristics are introduced for creating task scheduling orders when scheduling multiple workflows under user-specified deadlines, by applying deadlines either at the workflow level, at the task level, or in clusters of tasks across workflows. Fairness remains a primary concern in the optimization problem. However, the authors point out that assigning specific deadlines to groups of tasks within the workflows can lead to schedules that achieve a more efficient overall utilization of the available resources.

Deadlines can be used for prioritizing workflows (or part of them), depending on the urgency of their completion. The works of Malawski et al. [63, 64] focus on scheduling workflow ensembles in Cloud environments by prioritizing the individual workflows according to their user-defined importance. Ensembles are multi-workflows that share a similar structure and are characterized by different numbers of tasks and computational demands. Each workflow in the ensemble is assigned a priority level.

In detail, [63] formalizes the problem of scheduling ensemble workflows in Cloud environments under budget and deadline constraints. The proposed framework allows the submission of multiple workflows over time and uses an admission control mechanism to decide which workflow to accept or reject based on the available budget. Accepted workflows are scheduled using a heuristic algorithm that assigns tasks to resources by considering either workflow priorities, estimated execution times, or both.

In [64], the authors refine the optimization problem to maximize the number of completed workflows of an ensemble and formalize the workflow admission strategy according to the available budget. The study also considers task pri-

orities in multiple aspects of the scheduling process. The proposed algorithms exploit workflow structure and runtime estimates to ensure that admission and resource allocation decisions concentrate on the highest priority workflows that can finish within the budget constraint.

Genev et al. [65] also propose a methodology to schedule workflow ensembles in Cloud environments by prioritizing the individual workflows. Their main contribution lies in the flexibility of the scheduling approach, which enables easy switching between different optimization objectives without modifying the underlying algorithm.

To solve the optimization problem, a variant of the PSO meta-heuristic is proposed, where each candidate scheduling plan is represented by a particle which encodes task to resource mappings. In this encoding, the number of dimensions corresponds to the total number of tasks across all workflows in the ensemble. While PSO handles resource provisioning, prioritization comes into play when determining the scheduling order of workflow tasks. Priorities can be either assigned by users or determined according to different heuristics such as the upward rank metric used in the HEFT algorithm.

Zolfaghari and Abrishami [66] address the problem of scheduling workflow ensembles in the Cloud environment to maximize the number of completed workflows according to user-defined priorities under budget and deadline constraints. In particular, to reduce the overall cost of execution, spot instances are considered, that is, virtual machines offered at a lower price compared to on-demand instances, which can be interrupted by the Cloud provider when the demand for resources increases. The authors stress the importance of designing an admission strategy to reduce the number of partially completed workflows in the ensemble. The scoring function defined in [64] is used to evaluate the quality of scheduling plans and is refined to better capture the gain of completing lower-priority workflows as the number of workflows in the ensemble increases.

Scheduling is performed online to account for the fluctuating price of spot instances. The proposed heuristic is composed of four main phases: workflow acceptance, task scheduling order determination, instance provisioning, and fault tolerance. While acceptance and task order are primarily driven by workflow priorities and deadline constraints, the instance provisioning phase adjusts the number of virtual machines based on the number of ready tasks and the difference between the deadline and the start time of the task being allocated. Finally, a fault tolerance mechanism is integrated into the scheduling process to handle possible interruptions of spot instances, ensuring that tasks running

on interrupted instances are rescheduled on available resources to meet the deadline.

Chen et al. [67] extend [66] by proposing a heterogeneous Cloud model composed of on-demand and spot instances. The aim of the scheduling problem is to complete as many high-priority workflows as possible while minimizing the overall cost of execution. In particular, all workflows are constrained by a common deadline, while each workflow has a user-assigned priority level.

The proposed heuristic explicitly integrates spot instance price dynamics and availability into both resource provisioning and task assignment. This allows the scheduler to predict the execution time of individual tasks and trigger an ensemble-aware idle time slot exploitation strategy to ensure that the deadline is met. A dedicated spot monitoring mechanism continuously tracks spot instances availability from the Cloud provider, thus preventing task assignments to instances that are likely to be interrupted, especially from workflows with high priority.

Shafinezhad et al. [68] formulate a scheduling problem targeting Cloud resources and aiming at maximizing the number of completed high-priority workflows under budget and deadline constraints. The main contribution of this work is the design of a heuristic-based approach to solve the scheduling problem that combines different strategies. The fastest possible schedule is computed using HEFT for each workflow in the ensemble and is used to determine whether to accept or reject workflows based on the available budget and time before the deadline. For each workflow, tasks are scheduled, in the order established by HEFT, to the cheapest resources able to satisfy the deadline. The temporary workflow schedule is then evaluated to check for possible budget violations. If the budget is exceeded, the algorithm identifies idle time slots in the overall ensemble schedule and reassigns workflow tasks to cheaper resources within these gaps, thereby reducing the overall cost without affecting the deadline.

The rise of Fog computing, and more generally the Cloud-Edge Continuum, has opened research opportunities for multi-workflow scheduling in these environments. For instance, Siar and Izadi [69] address workflow ensemble scheduling in a distributed environment consisting of Cloud, Fog and IoT resources. The authors focus on offloading decisions and resource management to maximize the number of completed workflows under budget and deadline constraints. Building on prior work, they formulate the scheduling problem as a coalition-formation game in which groups of tasks and resources are formed

and released in an iterative process. Tasks join or leave coalitions according to their marginal contribution to the game's characteristic function, which captures the total group execution time and monetary cost. Although tasks are initially scheduled on IoT devices, each formed coalition is subsequently evaluated for potential offloading to Fog or Cloud resources to increase the overall number of completed workflows within the specified constraints. Constraints are enforced by accepting only workflows that can be completed within the specified budget and deadline.

3.1.4 SCHEDULING UNDER UNCERTAINTY

As discussed in Section 2.3, Cloud-Edge environments exhibit performance variability due to resource contention, virtualization overhead, and multi-tenancy, as documented in the survey [70], making task execution times and communication delays uncertain rather than deterministic.

This uncertainty directly affects workflow execution time and monetary cost [71], and undermines the effectiveness of scheduling algorithms that rely on fixed estimates of task execution times. Developing scheduling strategies that can effectively handle such variability is therefore a central concern in the literature.

The effects of performance uncertainty have been extensively surveyed in the context of Cloud environments [71, 72], and studied in other environments like network virtualization [73, 74]. Various approaches for handling uncertainty in workflow scheduling have emerged over the years, including stochastic provisioning models in Clouds [75, 76], uncertainty-aware workflow schedulers [77–84], multi-objective and fuzzy approaches in Fog–Cloud and Cloud–Edge systems [85–88].

For instance, in [75], uncertainty in application demands and resource performance is addressed through the formulation of a stochastic optimization problem aimed at minimizing the execution cost of applications in Cloud environments. The work integrates multiple sources of uncertainty in a unified cost-minimization framework over multiple Cloud providers.

To safeguard against demand uncertainty, the authors solve the scheduling problem offline by introducing a two-stage stochastic programming model. In the first stage, the model reserves resources ahead of time, while in the second stage, it allows for runtime adjustments based on the actual demand.

Meena et al. [76] formulate an optimization problem aimed at minimizing the execution cost of a workflow while meeting a deadline under variability in resource performance. Scheduling is performed offline using a GA that encodes both task-to-VM assignment and VM type selection. The major contribution lies in the fitness function used by the GA. A scalar parameter is used to model performance variability of Cloud resources. The nominal execution time of each task on a given VM type is inflated according to this parameter to capture the uncertainty in resource performance.

A different approach is taken by Della Vedova et al. [77], who propose a probabilistic formulation of the workflow scheduling problem. The work aims at minimizing the workflow expected monetary cost under a probabilistic deadline, thus ensuring that the workflow execution finishes within a given time with a specified probability. The Cloud infrastructure is modeled as a set of VM types with different cost and performance characteristics. The workload is represented by map and reduce tasks whose computational demands and data transfers are modeled as random variables following Log-normal distributions.

The authors design a heuristic that sorts VM types by increasing cost and, for each VM, greedily packs the maximum number of sequential tasks whose aggregated execution time satisfies the probabilistic constraint on the deadline. This process is iterated across VMs until all tasks are scheduled, and the first scheduling plan that satisfies the global deadline is selected.

Fard et al. [78] address workflow scheduling in Cloud environments under uncertain resource performance by modeling uncertainty in task execution times. The aim is to minimize workflow makespan and monetary cost, and to maximize robustness, defined as the ability of a schedule to cope with fluctuations in task execution times. The authors assume that task processing times cannot be characterized by precise values or even by known probability distributions. Instead, they assume that times lie within known intervals.

Robustness is defined at the workflow level and is quantified by a metric that incorporates the confidence interval of individual task execution times. The robustness of a schedule increases as the confidence interval of the execution time of the entire workflow becomes narrower. To solve the three-objective optimization problem, a heuristic based on MOHEFT is proposed.

Genev et al. [79] address the scheduling of workflow ensembles in Cloud environments aiming to minimize the overall makespan and monetary cost, and to ensure fairness of task allocation across workflows under user-defined deadlines. The authors model an environment where resource characteristics, and

particularly inter-Cloud bandwidth, are highly volatile. Uncertainty is associated with the bandwidth between resources, which is assumed to be known only through imprecise monitoring tools.

The authors propose a PSO-based scheduling algorithm that incorporates bandwidth uncertainty into the evaluation of candidate solutions. To this end, they introduce an estimate refinement mechanism that pessimistically scales bandwidth estimates according to a factor representing the level of uncertainty.

Uncertainty-aware scheduling has also been explored in online scenarios, where workflow arrivals and task execution times are only known at runtime. Chen et al. [80] study online scheduling for multiple workflows with the objectives of satisfying workflow deadlines in the presence of uncertainty in both task execution times and inter-task data transfer times, and of minimizing execution cost by improving resource utilization.

The proposed approach integrates proactive and reactive strategies within a single framework. In particular, task execution times and inter-task data transfer times are modeled as independent and normally distributed random variables. A key design choice is to limit the number of tasks waiting to be scheduled on each resource. Since task start, execution, and finish times are known only after its execution, restricting the number of waiting tasks on each VM limits how far uncertainty can propagate from predecessor to successor tasks. At the same time, the scheduler tries to fill idle gaps of each VM so that unused time is reduced while still satisfying workflow deadlines. The schedule is adapted at runtime to cope with deviations from expected task or transfer durations.

For handling uncertain task execution times in online multi-workflow scenarios, Liu et al. [81] investigate scheduling policies that simultaneously minimize VM leasing costs and satisfy workflow deadlines. The authors propose different deadline-assignment techniques (e.g., applied to individual tasks, task-batches, or workflow) in an online multi-workflow context, while addressing VM performance variability.

Each workflow has a user-defined deadline propagated from the workflow level down to the individual tasks so that every task is associated with a sub-deadline derived from the overall constraint. At runtime, the scheduler maintains a global view of all workflows and selects ready tasks across different DAGs, prioritizing tasks according to both their urgency regarding sub-deadlines and their structural importance (e.g., whether they lie on the critical path). Selection across workflows allows the algorithm to group tasks from multiple workflows onto the same resources whenever possible, thus

improving utilization while still satisfying all workflow deadlines. Crucially, the scheduling decisions are made under uncertainty in task execution times. Estimates of task duration and VM performance variability are incorporated into the priority calculation and VM selection logic.

An offline framework is presented in Calzarossa et al. [82], introducing a general stochastic approach for provisioning and scheduling data-parallel workflow DAGs, modeling VM performance via arbitrary distributions, and casting the scheduling problem as a single-objective cost minimization problem under a probabilistic deadline constraint.

After computing the expected execution times of the individual tasks, the approach computes the random variable representing the total workflow execution time. Similarly, the monetary cost of leasing Cloud resources is modeled as a random variable, since it depends on task durations.

The problem is solved using a Genetic Algorithm meta-heuristic which encodes provisioning and scheduling plans as individuals in a population and iteratively evolves them to minimize expected cost while satisfying a probabilistic deadline constraint.

Calzarossa et al. [83] investigate how the GA meta-heuristic design has to be adapted when fitness evaluation involves random variables. A sensitivity analysis of GA parameters (population size, selection pressure, and crossover/mutation probabilities) is performed, highlighting their interaction with uncertainty in affecting convergence and solution quality. The initial population is carefully constructed by combining bin-packing heuristics that take advantage of the task parallelism in the structure of the workflow DAG, with extreme solutions that focus on either minimizing makespan or monetary cost.

In [85], a multi-objective framework that simultaneously optimizes expected monetary cost and makespan under probabilistic constraints on both workflow deadline and budget is presented. The target infrastructure is a Cloud environment where workflow tasks can be mapped to heterogeneous resources. The proposed model supports arbitrary probability distributions to capture uncertainty in both workflow characteristics (i.e., task demands) and Cloud resources (i.e., processing capacity). Probabilities of meeting the deadline and budget constraints are integrated into the problem formulation.

The solution of the problem relies on a variant of the NSGA-II meta-heuristic, whose genetic operators have been adapted to handle both provisioning and

scheduling decisions. A Monte Carlo evaluation method is employed to estimate the expected cost and makespan of candidate solutions, as well as the probabilities of satisfying the given deadline and budget constraints.

Recent research has started to bring uncertainty-aware optimization into hybrid Fog–Cloud and Cloud–Edge settings. Rizvi et al. [87] address the problem of scheduling deadline-constrained scientific workflows across a hybrid Cloud-Fog environment to minimize their execution cost while meeting deadlines in the presence of uncertainty in task execution and communication times.

The authors propose a fuzzy scheduler that uses task features (i.e., computation time, communication overhead, deadline) to decide whether to offload each task to Fog or Cloud resources. Each feature is expressed in three linguistic terms (i.e., “small”, “medium” and “high”). To derive offloading rules, the resulting fuzzy input is processed using the Mamdani inference system [89]. For each task, execution can be performed either on Fog or Cloud resources, leading to a large search space of possible scheduling plans. To cope with this issue, the Salp Swarm meta-heuristic generates, evolves and learns the underlying Mamdani fuzzy rule base. Once the task-node allocation rules are learned, the same meta-heuristic is used to assign tasks to the Fog or the Cloud, and among the candidate resources, the one that satisfies the deadline with minimum execution cost is chosen.

Mokni et al. [88] employ multi-agent systems and fuzzy inference to reason about environmental uncertainties, while also simultaneously minimizing makespan and cost under budget and deadline constraints. Scheduling is addressed through a negotiation-based approach that balances conflicting optimization goals under uncertainty.

This framework partitions each workflow into latency-sensitive and compute-intensive tasks, offloaded to Fog and Cloud resources, respectively. Tasks are then sequenced to satisfy precedence constraints and scheduled by a multi-agent system in which Fog and Cloud agents negotiate via a fuzzy inference system that evaluates candidate schedules. Within each agent, a multi-objective GA searches for schedules over the resources it controls. The fuzzy logic layer supports decision-making when budget and deadline information is imprecise, while the distributed multi-agent system delegates scheduling decisions to the Fog and Cloud layers agents, thereby reducing global complexity and aligning decisions with the characteristics of the underlying resources.

Stavrinides and Karatza [84] study online scheduling of real-time IoT workflows in a three-tier IoT-Fog-Cloud architecture, where workflows arrive dynamically and task execution and communication times vary across invocations. The goal is to minimize deadline misses and improve resource utilization.

Workflows are submitted to the system according to a Poisson arrival process. The proposed scheduler distinguishes between compute-intensive/low-communication tasks and communication-intensive tasks, assigning the former to Cloud resources and the latter to Fog nodes to exploit the strengths of each layer.

In De Maio and Kimovski [86], a multi-objective workflow scheduler tailored to communication-intensive scientific applications that span Fog and Cloud is proposed. The goal is to jointly optimize response time, monetary cost, and reliability.

The NSGA-II meta-heuristic is employed to generate Pareto-optimal trade-offs across the three objectives. The scheduling model treats performance deterministically, without integrating uncertainty into the optimization process. In this work, a Monte Carlo engine is used to assess the robustness of the proposed framework by repeatedly simulating workflow executions over a Cloud-Fog infrastructure where hardware characteristics are drawn from predefined distributions modeling performance variability. By aggregating the results of such runs, the authors evaluate how consistently the scheduling decisions satisfy response time and cost objectives under variable environmental conditions, observing the advantages in using Fog resources for tasks with low computation and high data transfer needs.

3.2 *Autoscaling Techniques in Cloud Environments*

Autoscaling plays a vital role in ensuring performance, cost efficiency, and reliability of services in Cloud environments by adapting the amount of available computational resources in response to changes in incoming workloads.

In the literature, managing resource elasticity has been recognized as a central means to cope with the variability and unpredictability of workloads, and a rich body of works has been examined in a number of surveys [90–93]. In particular, methodologies are commonly classified on the basis of the mechanism and timing of scaling decisions into the three broad families introduced in Section 2.4, namely reactive, proactive and hybrid approaches.

The remainder of this section is structured according to these three approaches.

3.2.1 REACTIVE APPROACHES

Reactive autoscaling methods adjust the resources allocated to services in response to observed system metrics such as resource utilization, request queue length, or other Quality of Service (QoS) metrics.

As surveyed in [94–96], the works proposed in the literature span a wide spectrum of approaches of increasing complexity, ranging from simple static threshold rules and hysteresis controllers [97–100], through model-based schemes that encode cost, performance, and energy trade-offs [101–106], to fuzzy-logic, control theory and Reinforcement Learning methods [107–110].

Gandhi et al. [101] propose one of the earliest reactive approaches aimed at scaling data center resources. The approach continuously estimates the computational capacity needed to serve the current demand, while avoiding the measurement or forecasting of key workload characteristics such as request arrival rate, average request size, or overall system utilization. The autoscaling policy is periodically triggered and, at each invocation, inspects the current conditions of the system.

In particular, the total number of requests in the system is used as a feedback signal, and resources are provisioned to keep the per-server request count at a fixed target value, thereby maintaining a buffer of spare capacity to absorb sudden load increases without performance degradation.

Netto et al. [97] address the problem of satisfying QoS and budget constraints while avoiding systematic over-provisioning. The proposed approach aims at allocating the lowest number of resources to reduce runtime costs, while seeking high utilization of the allocated resources without violating QoS.

The amount of resources to be allocated or deallocated is computed according to their utilization level, and scale-out and scale-in actions are triggered when the utilization of the allocated resources reaches an upper threshold or is below a lower threshold, respectively. The authors evaluate the impact of the thresholds on the performance of the autoscaling policies, observing that the choice of the thresholds can significantly affect both cost and QoS.

Similarly, Calzarossa et al. [100] define a simulation-based framework for systematically comparing reactive autoscaling policies under diverse workload scenarios, including periodic patterns, steadily increasing request rates, and highly unpredictable arrival processes.

Two threshold-based policies are examined within this framework: the first monitors the utilization of individual VMs over time, while the second considers the aggregate behaviour of the allocated resources by tracking the fraction of busy VMs. In both cases, scaling actions are triggered when the computed utilization indicator crosses predefined thresholds, and a cooling time is enforced between consecutive actions to prevent oscillations. By systematically varying workload patterns, threshold values, and other configuration parameters such as the number of VMs allocated per scaling action, the authors evaluate how each factor influences completion time, VM utilization, and the frequency of scaling actions, highlighting the sensitivity of reactive policies to their operational settings.

In Liu et al. [107], thresholds are dynamically adapted according to observed workload, response time, and the CPU load of the application resources running in Cloud environments. Rather than relying solely on precise calculation, a fuzzy control model is employed to translate measurements of highly variable workloads and noisy monitoring data into timely and appropriate scaling actions.

A central aspect of the fuzzy model is the use of dynamic upper thresholds for response time, since the threshold should depend on the amount of allocated resources. When a larger number of resources is being allocated, a moderate change in incoming load tends to produce only small variations in response time. Under these conditions, the fuzzy rules drive the controller to permit a higher upper response time threshold before additional resources are allocated, thereby avoiding unnecessary scaling out. In contrast, when few resources are

allocated, even a slight increase in request rate can lead to substantial response time fluctuations and a high risk of violating QoS. In these conditions, the fuzzy engine uses a much tighter upper threshold, triggering scale-out earlier.

The increased usage of containerized applications has led to an increased interest in reactive autoscaling approaches specifically designed for Kubernetes [111] and similar container orchestration platforms [112]. For instance, Casalicchio and Perciballi [104] aim at improving the Kubernetes Horizontal Pod Autoscaler (KHPA) by leveraging estimates of absolute CPU utilization derived from host-level measurements. The authors argue that traditional KHPA relies on relative metrics (e.g., CPU utilization percentage), which can lead to imprecise scaling decisions when the relation between these indicators and the actual resource demand is not well captured at run time.

The proposed autoscaling algorithm bases its decisions on the absolute CPU demand learned from measurements on a real container-based system. The approach consists of two main phases: the first phase is an offline calibration step aimed at deriving a correlation model linking relative metrics to absolute resource needs for the specific application and environment. In the second phase, relative measurements are converted at runtime into absolute metrics using the derived correlation model, and scaling decisions are made based on these absolute values. A follow-up study [105] finds that the relationship between relative utilization and absolute demand is highly workload-specific, motivating periodic re-estimation of the correlation model at runtime to track changing workload characteristics.

The delay introduced by resource boot and decommissioning latency is a fundamental challenge for reactive autoscaling, as it can lead to oscillations and performance degradation if not properly accounted for. In this regard, Quattrocchi et al. [108] address this challenge directly, observing that the autoscaling control period is fundamentally bounded by actuation latency, which ranges from approximately one minute for VM provisioning to a few seconds for container startup, and setting the period of each controller to match this latency so that a new scaling decision is only issued once the previous one has taken full effect. Autoscaling is formulated as a control problem using a control loop with a defined period constrained by the time needed to add or remove resources, with two complementary controllers being proposed.

The first is a hierarchical control-theoretic autoscaler that operates primarily on container-based applications. It employs proportional–integral (PI) control loops associated with individual containers to continuously regulate allocated resources against response time or deadline-based objectives. It is designed to

run fast control loops while still respecting actuation delays, thus reducing the risk of instability and resource oscillations.

The second controller follows a model-based control approach, embedding a queuing network model of the application within the controller and repeatedly solving an optimization problem to determine the minimal resource allocation that satisfies performance targets under current workload conditions.

As public Cloud offers a wide variety of resource types with different performance characteristics and costs, some works have focused on reactive autoscaling that can select among heterogeneous resource types to optimize cost and performance. For instance, Iqbal et al. [98] propose a strategy for multi-tier web applications hosted on Amazon EC2, with the specific goal of minimizing cost while ensuring that the 95th percentile of response time remains below a target threshold.

The benchmark application is organized into two independently-scaled tiers (i.e., a web tier and a database tier), each backed by a set of VMs. Two rule-based scale-out policies are defined: the first scales-out a tier when its average VM CPU utilization exceeds a configured threshold, while the second triggers on response time target violations and identifies which tier to scale-out based on the same utilization threshold. The CPU utilization threshold is tuned per EC2 instance type through pilot experiments to identify the maximum sustainable load per-VM without degrading performance.

Reinforcement Learning (RL) is applied in the autoscaling policies proposed by Russo Russo et al. [109] for data stream processing applications deployed on heterogeneous Edge, Fog, and Cloud resources. The policies aim at satisfying response time constraints and minimizing monetary cost by automatically selecting the most appropriate resource type (i.e., Edge, Fog, Cloud) to allocate.

A two-layer hierarchical control architecture is proposed, in which an upper application-level manager coordinates a set of lower operator-level managers, each responsible for autoscaling a single operator of the application. Each operator-level manager is implemented as an RL agent that solves a local Markov Decision Process, deciding the number of operator replicas and the resource type on which they run so as to satisfy a per-operator response time target while minimizing cost and reconfiguration overheads. The application-level manager uses Bayesian Optimization to build an approximate performance model of the application at runtime, and exploits it to translate the user-provided QoS requirement into the per-operator targets that the

lower-level agents must satisfy. In this way, users are relieved from manually tuning low-level control parameters such as utilization thresholds.

Reinforcement learning is also employed by Feng et al. [110] in a vertical scaling context, aiming to minimize the sum of infrastructure leasing costs and Service Level Agreement (SLA) violation penalties for database containers. The authors consider a Cloud environment in which scaling decisions must cope with high workload variability while avoiding the high startup and reconfiguration overheads characteristic of vertical scaling.

To this end, they propose an RL-based vertical scaler with a cost-minimization reward formulation and a dynamic action model: in contrast to previous RL autoscalers, which map each action to a fixed resource adjustment, the proposed actions select among qualitative adjustment levels (e.g., marginal, moderate, or substantial scale-up or scale-down) whose numerical magnitudes are recomputed at each step from the estimated resource demand.

To conclude, it can be observed that the effectiveness of reactive autoscaling is bounded by the quality of the information available at decision time. Since scaling actions are issued in response to a measured performance deviation, autoscaling approaches cannot anticipate sharp load transitions and can only intervene on conditions that have already occurred. The behavior of reactive autoscaling also depends strongly on configuration parameters such as thresholds, control gains, or reward signals, whose calibration is workload-specific and must be revised whenever the operating environment changes. Adaptive schemes based on fuzzy logic, control theory, or reinforcement learning reduce this dependence but cannot remove it, since the mechanism that updates them at runtime itself relies on assumptions about the workload.

3.2.2 PROACTIVE APPROACHES

Proactive autoscaling approaches have been introduced to address the inherent limitations of reactive mechanisms [113], which can struggle to maintain performance and cost-effectiveness under rapidly changing workloads.

Proactive strategies leverage predictive models to forecast future workload demands, thus triggering scaling actions before demand exceeds capacity.

Proactive autoscaling has been extensively surveyed in [114–117] and studied from a modeling perspective in [118]. Many works building on this idea

have emerged, spanning classical time series and statistical forecasting techniques [119–123], machine and deep learning predictors [113, 124–136], and probabilistic and optimisation-driven controllers encoding uncertainty, cost, and system dynamics [137–143].

Early works introduce proactive resource estimation into reactive frameworks, relying on lightweight statistical summaries of recent workload history to anticipate near-future demand rather than reacting to instantaneous metrics alone. Sahni and Vidyarthi [144], for instance, design a heterogeneity-aware autoscaling mechanism that captures the main advantages of both reactive and proactive techniques, while avoiding the overhead and potential inaccuracy associated with prediction models. To this end, they propose a reactive approach that, while relying on simple, rule-driven scaling decisions, is aware of the necessity of estimating future resource needs.

Scale-in and scale-out decisions are triggered when the utilization of the allocated VMs reaches an upper threshold or is below a lower threshold, respectively. Moreover, the algorithm uses a moving-average-based model and a standard-deviation-based safety margin to approximate the expected change in incoming traffic, and uses an analytical queuing model of the allocated VMs and response time constraints to compute the required capacity.

The algorithm takes into account the price and performance differences of available VM types to add capacity in a cost-optimized way, while maintaining the QoS objectives even in the presence of varying workload.

The autoscaling policy proposed by Roy et al. [119] is an example where workload forecasting is combined with a simple look-ahead optimization mechanism that plans how many resources should be allocated over a short time horizon, balancing machine leasing costs against the risk and penalties of violating SLAs. The authors observe that, in theory, the optimal strategy would be to adjust resources continuously as workload fluctuates, reducing the time interval between reallocations to zero. In practice, however, each reallocation incurs non-trivial overhead from VM boot times, configuration delays, and state transfers, making such continuous adjustment infeasible. The policy therefore operates over a limited set of discrete time intervals, and incorporates a re-configuration cost term in its objective function that acts as an inertia penalty, discouraging unnecessary thrashing between scale-in and scale-out actions.

At each step within the planning horizon, the approach first predicts the incoming workload using a second order ARMA model. For each possible resource allocation, the approach estimates the resulting performance and associated cost using Mean Value Analysis. By iteratively evaluating all candidate

allocations, it builds a cost profile over the entire horizon. Finally, the method formulates and solves a single-objective optimization problem to select the resource provisioning plan that yields the minimum total cost, accounting for both infrastructure cost and SLA-related penalties.

Gandhi et al. [137] introduce a model-driven autoscaling framework designed to satisfy application performance guarantees while keeping infrastructure costs low. Multi-tier Cloud environments are considered, where applications are decomposed into separate layers, such as a web front end, an application logic tier, and a database back end, each potentially hosted on different VMs. The proposed framework is built around three main components: a monitoring agent, a modeling engine, and an execution engine integrated with OpenStack. Each time the monitoring agent is triggered, it gathers measurements such as CPU utilization from the VM hypervisors, together with the incoming request arrival rate.

In the modeling engine, an Extended Kalman filter leverages these measurements to infer unobservable parameters such as service times at each tier, network delays, and background utilization caused by external load not related to the monitored application. The system is modeled as a product-form queueing network, which admits closed-form expressions linking per-tier utilization and end-to-end response time to the number of VMs per tier, the service demands, and the arrival rates. These expressions, evaluated with the parameter estimates produced by the Kalman filter, are solved to determine, for every tier, the minimum number of VMs needed to keep the overall response time within the desired QoS level. Finally, the execution engine issues scale-in and scale-out directives to OpenStack only after two consecutive consistent recommendations from the modeling engine, avoiding reactions to transient fluctuations in the estimates of the Kalman filter.

Abdullah et al. [142] address autoscaling over heterogeneous VM types, where each scaling decision must specify both how many resources to add or remove and which types to provision, so that response time constraints are satisfied at minimum cost. To solve this problem, the authors propose a reinforcement learning-based autoscaling framework that integrates VM type selection into the scaling loop, under the constraint of a fixed target average response time and a finite set of available VM types.

The framework follows an exploration-exploitation paradigm. An initial exploration phase applies reactive scaling with randomly chosen VM-type mixes to build a “capacity map” that records, for each configuration, the maximum request rate sustainable within the response time target. During exploitation,

the reinforcement learning agent operates on a Markov Decision Process where each state captures the current request arrival rate and the per-type count of allocated VMs, and each action specifies the per-type change in VM count, i.e., how many resources of each type to add or remove. A K-Nearest Neighbor serves as a predictor for the Markov Process, guiding scaling actions to keep average response time under the specified service level objectives while minimizing monetary cost.

Other works [120, 124, 127, 129] move beyond single-model predictors and fixed decision rules, combining or switching between forecasting strategies at runtime and adapting the scaling policy to the observed workload regime, following the taxonomy of self-adaptive autoscaling systems surveyed in [145].

For instance, Gong et al. [120] propose a framework that operates by continuously monitoring resource utilization and feeding measurements into two different chosen predictors depending on the detected workload pattern. The aim is to detect and exploit possible patterns in the incoming workload to perform short-term forecasts of the demand. The framework is built upon two short-term predictors: a signature-based predictor and a state-based predictor.

The signature-based predictor assumes a periodic pattern in the incoming workload. To uncover such regularities, a Fast Fourier Transform is applied to the recent history of resource usage and the dominant frequencies in the signal are identified. These frequencies represent recurring patterns in demand, which are then used to forecast resource needs. When no periodic pattern is detected, a state-based predictor is used. The range of resource usage is discretized into equal-width bins, each corresponding to a state of a discrete-time Markov chain, and the transition probabilities between states are estimated from the observed history. Forecasts are obtained by propagating the current state forward through the transition matrix and selecting the most probable state at the desired horizon.

The objective of Nikraves et al. [124] is to improve the accuracy of predictive autoscaling by selecting, at runtime, the most appropriate time series prediction algorithm as a function of the current workload pattern.

A self-adaptive prediction suite is introduced, which incorporates several time-series predictors, namely Support Vector Machines (SVMs) and two versions of Multi-Layer Perceptrons (MLPs), differing in their risk minimization principle (Empirical Risk Minimization vs. Structural Risk Minimization). Learning-theoretic analysis is used to reason about which class of model should be preferred for workloads of differing complexity. The suite implements

an autonomic control loop and is designed using the Strategy and Template design patterns: a workload context autonomically classifies incoming workload (periodic, growing, unpredictable) via time series decomposition (Loess) and passes a pattern interface to a predictor component, which selects and configures the appropriate regression model and sliding-window size according to a set of fusion rules derived from the authors' theoretical and empirical investigation.

The idea of model selection has been extended by Alidoost Alanagh et al. [127] to consider more diverse workloads and a richer predictor portfolio. The objective is to address the problem of Cloud elasticity for heterogeneous workloads where a single prediction model cannot capture the range of temporal behaviors and naïve ensembles are too slow for online autoscaling decisions. To this aim, they propose an adaptive autoscaling architecture in which a neural network classifier routes each incoming job to the most suitable prediction algorithm at runtime, based on sequential statistical features extracted from the workload history.

The architecture extends a control loop with an admission controller, a load balancer, a configuration tool, a workload monitor, and a workload predictor; the latter operates in 5–10 minute control cycles, determining the number of virtual machines to be added or removed. In detail, the neural network model is trained offline on workload samples to learn which predictor is best for each workload type using a cost function based on the Mean Relative Prediction Error. At runtime, the predictor component uses this model to select, for new workload instances, the most suitable forecasting algorithm among Linear Regression, Support Vector Machines and ARIMA. The chosen predictor then forecasts near-future CPU demand to drive horizontal scaling of VMs.

Chouliaras and Sotiriadis [129] propose an adaptive autoscaling framework for Cloud resource provisioning that addresses the limitations of static, threshold-based scaling rules: being manually configured and fixed, such rules cannot adapt to fluctuating workload demand, leading to either QoS violations or resource over-provisioning, in both single- and multi-tenant environments. The main contribution lies in a provisioning method that models workload and system behavior and uses these models to guide vertical scaling decisions on the containers, raising or lowering the CPU allocation of each container. In particular, the historical CPU utilization time series is first segmented into fixed-length sequences, and K-means clustering is then applied to group these sequences into three representative demand states: High, Medium, and Low.

For workload prediction, a 1-D CNN is trained on historical CPU utilization

data to forecast future usage values; the resulting forecast is then segmented using the same fixed-length window, and each segment is labeled by matching it to the nearest representative cluster, yielding a predicted sequence of High, Medium, and Low demand states. The resulting sequence of demand states is mapped to vertical scaling actions, which a deployment manager applies only if the target configuration is available and its projected cost remains below a user-specified monetary cap. The authors explicitly acknowledge that the approach assumes repeatable workload patterns, which limits its applicability to non-repeatable or highly irregular workloads.

Some approaches [122, 128, 130, 134–136] focus on burstiness detection, that is the ability to identify and react to sudden, short-term spikes in demand that can cause performance degradation if not handled proactively.

In particular, Tahir et al. [122] forecast incoming workload using a Decision Tree regressor trained on workload traces from simulations. Burstiness is detected over a short history of recent workload observations using one of three techniques: Sample Entropy, Normalized Entropy, or Fast Fourier Transform. The autoscaling algorithm combines the burst decision with a short history of predicted resource counts to decide scale-in or scale-out, notably preventing scale-in and allocating the maximum predicted resource count during detected bursts, thereby prioritizing QoS protection over cost during these critical periods.

A different framing is adopted by Abdullah et al. [128], who use various regression models to perform workload forecasting and to predict the number of required resources according to the forecasted workload and the maximum admissible response time. To detect burstiness, the autoscaler maintains a buffer of recent resource count predictions. The variance of these predictions is computed over sub-windows of increasing length ending at the current step. If the variance within any sub-window exceeds a predefined threshold, the autoscaler enters a burst mode and allocates the maximum predicted resource count within that sub-window.

Vu et al. [134] place a Bidirectional LSTM forecaster at the core of a proactive autoscaling loop for Kubernetes, with a burst detector that flags sudden spikes from the moving average and standard deviation of the *predicted* arrival rate rather than from currently observed metrics. Incoming load is forecasted at each control interval, and a multi-output Decision Tree Regressor maps the forecast, a candidate per-pod CPU allocation, and the target response time to the number of pods required and the resulting per-pod utilisation. Under

regular operating conditions, both the number of pods and the per-pod CPU allocation are adjusted from this mapping so that the response time constraint is satisfied while utilisation is kept high. When the burst detector fires, the controller instead allocates the maximum per-pod CPU and restricts scaling to the horizontal dimension, trading cost-efficiency for QoS protection during the spike. Since vertical adjustments in Kubernetes generally require recreating pods, a rolling update controller applies scaling decisions by creating a new deployment with the selected configuration alongside the old one until the transition completes.

A similar Bi-LSTM-driven autoscaler for containerised applications is proposed by Mogal and Sonaje [135], differing mainly in the burst identification rule. Rather than relying on the dispersion of the predicted workload, the burst state is entered whenever the predicted arrival rate crosses a configurable threshold and is only gradually released through a decay function, which dampens sensitivity to short transient overshoots in the forecast. A detected burst pins the per-pod CPU at its maximum and restricts scaling to the horizontal dimension, while normal operation jointly adjusts replica count and per-pod CPU from the forecast.

In the smart-campus setting considered by Razzaq et al. [136], an ensemble workload predictor drives pre-scaling across both the number of running resources and their per-resource CPU allocation. Burst handling is folded into the same prediction-driven policy, which prioritises sustained availability when burstiness is detected on the forecast and otherwise optimises for cost-efficiency. Relative to the other burst-aware works, the distinguishing feature is the ensemble-based predictor paired with a deployment-specific cost model rather than a single deep forecaster with a dedicated detector module.

Some other works [121, 126, 131, 132, 143] focus on Deep Learning-based proactive autoscalers, which exploit neural predictors to infer workload patterns and cope with bursty workloads in Cloud-native environments.

For instance, the predictive model presented by Kumar and Singh [121] leverages neural networks to learn and extract patterns from the workload to be used for future predictions. The authors propose a three-layer feedforward neural network that, at every autoscaling step, receives as input the last measurements of the request arrival rate.

The model is trained in a supervised manner using Mean Squared Error as the loss function. Rather than relying on conventional gradient-based back-

propagation, the authors employ a Self-adaptive Differential Evolution algorithm to optimize the synaptic weights. In this evolutionary setup, a population of candidate neural networks is iteratively improved using different mutation strategies. The mutation operator is selected at each step through a roulette-wheel mechanism, and both the mutation factor and the crossover rate are continuously updated based on the observed success of generated offspring. In this way, the method simultaneously tunes the network weights and the evolution hyperparameters. Once trained, the predictor is embedded into a Cloud resource management system. Its forecasts of near-future workload, combined with the current state of the infrastructure, are then used to make proactive autoscaling decisions.

Zhou et al. [126] address the problem of response time service level violations in containerized microservices subject to highly dynamic and bursty workloads. The authors propose a proactive resource provisioning framework that pairs a buffer of pre-initialized warm containers, which absorb sudden request surges without incurring cold start delays, with a predictive autoscaler that adjusts the number of active container replicas to track forecast demand. The approach assumes microservices run as multiple container replicas in a Kubernetes cluster, partitioned into work containers that serve incoming requests and warm containers that remain idle but ready for activation, with the fraction of warm capacity bounded by a configurable upper limit. The framework comprises three components: a workload predictor, a microservice performance model, and a controller that orchestrates scheduling and scaling decisions.

The workload predictor is an LSTM neural network that, given a sliding window of recent arrival rate observations, forecasts the near-future values. The microservice performance model is built from profiling traces collected via an automated build and testing pipeline (Continuous Integration and Delivery, CI/CD), in which each microservice is stress-tested using the ‘wrk’ HTTP benchmarking tool under a simple threshold-based reactive autoscaler.

From these traces, two models are learned: an XGBoost regressor that predicts response time as a function of incoming workload and allocated container count, and a linear regressor that estimates cold start delay from the current container count and the number of containers to be added.

The controller runs a proactive algorithm at fixed intervals, using response time predictions to adjust the number of work containers and cold start predictions to size the warm container buffer. When predicted demand exceeds current work container capacity, incoming requests are immediately redirected to warm containers, providing a fast fine-grained response, while the autoscaler adjusts the work container count on a coarser timescale.

The work of Theodoropoulos et al. [143] introduces an autoscaling framework that pursues two objectives jointly: improving the responsiveness of the managed applications through proactive horizontal scaling driven by multi-step forecasts, and minimizing the resource overhead introduced by the autoscaling logic itself. To meet the second objective, the authors adopt a single global forecasting model that captures structural relations between resources, rather than running a specialized predictor per resource. The autoscaler is organized as a four-stage pipeline comprising a Monitoring Component, an Input Construction Component, a Forecasting Component, and an Output Distillation Component.

The Monitoring Component, built on standard observability tools (Prometheus [146] and kube-state-metrics [147]), periodically collects resource-level CPU utilization along with static metadata such as the cluster each resource belongs to and the application it implements. The Input Construction Component uses this metadata to build a similarity graph over resources, in which each Edge weight reflects how strongly two resources are expected to behave alike: zero for resources of different applications (treated as independent), an intermediate value for resources of the same application in different clusters, and the highest value for resources of the same application within the same cluster. Attaching the recent CPU measurements to the nodes of this graph yields a sequence of graph snapshots with fixed topology and time-varying node features.

The Forecasting Component processes this sequence through an encoder-decoder model in which a weighted Graph Convolutional Network encoder captures the dependencies induced by the graph structure and an LSTM-based decoder models their temporal evolution. A single forward pass emits multi-step CPU utilization forecasts for every resource at once. Finally, the Output Distillation Component converts these multi-step forecasts into a single target replica count per resource. For each resource, it selects the forecast at the horizon just beyond the boot time of the resource: since a newly provisioned replica becomes operational only after its boot delay elapses, issuing the scaling command on the basis of that forecast aligns its readiness with the moment the predicted demand materializes.

Various approaches [139–141] take into account the high variability in performance capabilities of Cloud environments [148], and explicitly account for workload and performance uncertainty by using probabilistic forecasting or stochastic models to derive robust autoscaling policies.

Regarding uncertainty in workload arrivals, Qian et al. [139] address autoscaling under uncertain and highly variable workloads in a “scale-per-query” setting, which is common in Function-as-a-Service (FaaS) and CI/CD systems. In this model, every incoming request spawns its own short-lived resource, which is then terminated immediately after processing that single request. Because creating these resources can introduce noticeable cold-start delays, the goal is to proactively allocate resources so that QoS targets are met without overprovisioning.

To achieve this, the authors introduce a QoS-aware autoscaling framework composed of four components: periodicity detection, historical query arrival modeling, query arrival prediction, and scaling plan module. The core of the approach lies in the historical query arrival modeling and prediction parts. Request arrivals are modeled as a non-homogeneous Poisson process, which is then used to project the timing and intensity of future requests in the short term.

The autoscaling problem is formulated as a stochastic optimization problem whose objective is to derive the number and timing of resource creations that trade off the expected provisioning cost against the probability of meeting QoS targets, since under the stochastic arrival model both quantities depend on the random sequence of incoming requests. The optimization is solved via Monte Carlo “sort-and-search” techniques. A proactive scaling algorithm then uses these optimization outputs to schedule resource creations ahead of time via a sequential decision process, where each scaling action accounts for the resources already consumed by prior query arrivals, ensuring that queries are served with high probability without incurring excessive cold-start delays.

Hang et al. [141] address the robustness limitations of predictive autoscaling techniques in Cloud databases. The authors design a predictive autoscaling framework able to quantify forecast uncertainty while keeping resource usage efficient. The approach comprises a probabilistic workload forecaster and an autoscaling manager.

The forecaster takes historical workload time series as input and produces quantile forecasts using deep learning models, either by learning parametric distributions from which quantiles are sampled (e.g., autoregressive recurrent networks) or by directly predicting a fixed grid of quantiles (e.g., temporal fusion transformer).

The autoscaling manager formulates an optimization problem over a future horizon, aimed at minimizing the total number of compute nodes while limiting the average workload per node to stay below a threshold derived from the forecasted quantiles. The quantile level used to derive this threshold is selected

adaptively at each step: an uncertainty metric, computed as the weighted deviation of the quantile forecasts from the median, is compared against a configurable threshold, and the autoscaler applies a conservative (higher) quantile when uncertainty is high and an aggressive (lower) quantile when uncertainty is low.

Pan et al. [140] propose an uncertainty-aware predictive autoscaling framework for multi-tenant Cloud clusters, in which shared resources serve the aggregate demand of all tenants and that demand is volatile. Given a history of the aggregate demand, the goal is to decide how many resources to launch or drain at each future step so as to minimize resource-related costs while bounding the risk of violating QoS requirements. The framework is split into a predictor and a scaler.

The predictor combines a Multi-scale Attentive Feature Extractor, which captures both short-range fluctuations and long-range trends in past demand, with a Gaussian Process Regression model that maps the extracted features to a probability distribution over demand at each future step.

The scaler is formulated as a Markov Decision Process whose state comprises the number of running resources and the accumulated unmet demand, i.e., the backlog of requests that earlier provisioning decisions could not serve, and whose actions specify the number of resources to launch or drain at the current step. The cost function combines the expected idle-capacity cost, computed under the forecast distribution, with the cost of each scaling action; QoS is enforced through a constraint on expected unmet demand, governed by a tolerance factor that lets operators trade off cost against allowed SLA violations.

3.2.3 HYBRID APPROACHES

Hybrid autoscaling approaches combine proactive and reactive mechanisms. The specific integration pattern varies considerably across the literature. In some works, the two components govern different scaling directions, with a reactive policy responsible for one direction and a proactive predictive model for the other. In other works, a proactive forecasting component anticipates future demand and drives provisioning decisions, while a reactive mechanism compensates for workload deviations that the forecasts fail to capture.

The first of these patterns is illustrated by [149–151], where scale-out and scale-in are governed by explicitly distinct decision logics. In some cases, one scaling direction relies on threshold breaches in currently observed metrics while the

other is informed by a predictive model of future load or resource utilization; in others, a single predictive mechanism governs both directions under deliberately asymmetric decision rules.

In particular, Iqbal et al. [149] use a reactive heuristic to identify bottlenecks and trigger scale-out decisions, while an online regression model predicts over-provisioning conditions and triggers scale-in decisions. The work addresses the problem of satisfying response time SLAs for highly variable workloads in two-tiered Cloud applications consisting of a web server tier and a read-only database tier, where the interactions between tiers are not directly observable.

A reactive heuristic triggers scale-out decisions on the basis of the application logs and a threshold placed on the 95th percentile of response time. Scale-outs target the web server tier when static content is slow or web server utilization is saturated, and the database tier when only dynamic content is slow while web server utilization remains below threshold.

Regarding scale-in actions, an online quadratic regression model predicts the required number of web and database resources as a function of observed static and dynamic request rates. The model is updated incrementally using only the intervals in which the SLA is satisfied. When the model indicates that the current configuration is over-provisioned, VMs are removed from the corresponding tier.

The work of Bouabdallah et al. [150] combines a reactive scale-out mechanism with a proactive scale-in approach. The work addresses the problem of scaling of web resources in Cloud environments, with the objective of preventing resource saturation and reducing over-provisioning.

The proposed methodology is based on a multi-agent system following a MAPE-K loop where the monitoring agent periodically measures CPU usage and sends it to the analyzer agent. In this agent, a threshold-based reactive rule is applied to trigger scale-out actions, while a proactive component is used to decide scale-in actions.

The proactive component uses Simple Exponential Smoothing over a sliding window of recent observations of CPU usage, selecting the smoothing factor that minimizes the mean squared error on past data, and requests a scale-in when the forecasted CPU utilization falls below a user-defined under-utilization threshold. A manager agent plans and executes the resulting scaling actions against the provisioning API of the Cloud provider, consulting a Cloud registry that acts as the shared knowledge base recording the current resource inventory.

Ali-Eldin et al. [151] generalize the idea of autoscaling by introducing a formal “elasticity engine” with separate scale-in and scale-out controllers, each of which may be configured as either reactive or proactive. The work addresses the challenge of handling flash crowds in Cloud-hosted services, where sudden traffic surges risk causing SLA violations before reactive adjustments take effect, and where premature scale-in decisions induce resource oscillations that amplify subsequent under-provisioning episodes.

Autoscaling is formulated as a control-theoretic problem where the Cloud system is abstracted as a G/G/N queue with a variable number of available servers in which both arrival and service times follow a generic probability distribution. On top of this model, the autoscaling mechanism is described as a closed-loop control system. The elasticity controller at the core of the autoscaler takes as input the current request arrival rate and the measured service throughput, and produces as output the number of VMs to add or remove at each time step. The control objective is for the measured throughput to track the total demand, modeled as the arrival rate plus an unpredictable fluctuation component, within an error tolerance specified by the SLA.

By instantiating the scale-out and scale-in controllers with different combinations of reactive and proactive strategies, the framework yields a family of hybrid controllers that blend the two behaviors asymmetrically across scaling directions.

Other works [152–159] combine the reactive and proactive components in a single control loop, where forecasts and measurements are combined at every control step to determine the next scaling action.

For example, Urgaonkar et al. [152] investigate the problem of dynamically provisioning capacity for multi-tier applications hosted on server farms, where workloads exhibit both bursty and long-term seasonal patterns. The goal is to allocate resources across application tiers so that end-to-end response time SLAs are met even under bursty arrivals and sustained high demand.

The proposed approach models each multi-tier application as a network of G/G/1 queues and uses nominal per-tier request service time and online measurements of interarrival characteristics to compute, for a given peak session, the average response time and throughput. A proactive component maintains, for each hour of the day, a histogram of arrival rates observed during that hour across the preceding days, and, at the start of each hour, provisions capacity sufficient to serve the arrival rate corresponding to a high percentile (e.g., 95th) of that histogram. A reactive component monitors the ratio of observed

to predicted load, counts requests with violations, and, when thresholds are exceeded, increases capacity at saturated tiers.

In the hybrid policy proposed by Singh et al. [155], the proactive component forecasts the peak request arrival rate in the next scaling interval, whereas the reactive component considers recent CPU utilization and response times with the aim to provide robust, cost-efficient autoscaling for multi-tier web applications running on IaaS Clouds under highly variable workloads.

The autoscaling approach is structured around a MAPE-K control loop. The monitoring phase collects incoming request rate, per-VM CPU utilization, and response times, aggregating them into averaged metrics for each interval. In the analysis phase, a proactive time series forecaster is applied to a sliding window of recent request arrival rates, and then selects and applies ARIMA, linear regression, or support vector regression with seasonal post-processing to predict the maximum incoming workload in the next scaling interval. Simultaneously, the reactive component averages CPU utilization and response time over the last minute.

In the planning phase, the reactive and proactive signals are combined sequentially into a single scaling decision. Concurrent violations of the upper CPU utilization and response time thresholds trigger immediate scale-out, whereas concurrent low values of both metrics mark the interval as a candidate for scale-in. Whenever the reactive rules do not already commit to a scale-out, a proactive capacity planner applies an M/M/m queueing model to the predicted workload, the per-VM throughput, and the SLA response time target to derive the VM count required to meet the target; if this exceeds the in-service and booting VMs, the reactive recommendation is upgraded to a scale-out.

The execution phase enforces the planning phase decisions by randomly selecting VMs to scale within the allocated resources.

Rampérez et al. [156] target Cloud deployments in which the bottleneck resource is not known a priori and only low-level resource measurements are available. In this setting, the proposed hybrid autoscaling framework addresses the three runtime elasticity decisions (i.e., when to scale, which resource to scale, and by how much). The proposed approach continuously monitors the workload and resource utilization to learn predictive models that capture how provisioning actions affect SLA behavior. These models are used at runtime to forecast both future SLA and the time required for scaling actions.

The framework is organized in four components. A scaling time forecaster uses a linear regression model to estimate the time interval between autoscaler

invocations as a function of current workload. A workload trend forecaster computes time-series predictions of future workload evolution over a given prediction horizon. A performance forecaster collects a set of resource metrics and learns regression models mapping them to response time, thereby identifying the performance indicators most relevant to autoscaling, as well as potentially saturated resources.

A decider component evaluates the predicted response time trend from the Workload Trend Forecaster against configurable thresholds to determine whether to scale proactively. As a contingency against forecast errors, it also checks current performance estimates against separate reactive thresholds. A cool-down period is enforced after each scaling operation to prevent oscillations.

Joshi et al. [157] aims at improving average response times and resource utilization with respect to KHPA by proposing a hybrid framework that combines a proactive time series predictor with a reactive feedback controller. The proactive part consists of an ARIMA model, which forecasts short-term CPU utilization from historical metrics and maps the prediction to a desired number of containers to provision. The reactive part is a Proportional–Integral–Derivative (PID) controller, which monitors actual CPU utilization and derives the desired container count to reach a utilization level of 75% for all containers.

The final desired container count is computed as the weighted average of the outputs from the reactive and proactive components. Weights are tuned empirically, e.g., by giving a larger weight to ARIMA for anticipation and a smaller weight to PID for stabilizing short-lived bursts, allowing the relative contribution of proactive and reactive behavior to be calibrated to the workload characteristics of the deployment.

Zou et al. [158] propose a stochastic, hybrid autoscaling framework for long-running applications that must handle highly variable workloads while meeting strict SLA constraints. The system automatically adjusts the number of resources so that aggregate CPU utilization remains close to, but below, a target level, while enforcing limits such as minimum and maximum resource counts, bounds on how quickly resources can be added or removed, and uncertainty due to workload bursts, measurement noise, and interference between co-located applications.

The framework consists of three tightly integrated components. A proactive component is in charge of performing multi-step-ahead workload forecasting by combining a long-term periodic predictor with a short-term local predictor whose outputs are fused into a single workload prediction. The long-term

model uses a Fourier series decomposition to capture regular workload patterns, while the short-term predictor focuses on recent changes and anomalies. Training employs a quantile loss function, so that the model estimates a high quantile of the workload distribution rather than its mean, producing deliberately conservative forecasts that reduce the risk of under-provisioning.

The reactive component estimates the relationship between the workload of each co-located application and the aggregate CPU utilization observed at the cluster level. A linear model is initialized via maximum-likelihood estimation on historical data and continuously refined through linear regression as new measurements become available. The optimization component converts the outputs of the proactive and reactive modules into scaling actions. Autoscaling is cast as a Model Predictive Control problem that aims to maximize CPU utilization over a finite prediction horizon, subject to constraints on scaling rate and total resource count. At each control step, a nonlinear optimization problem is solved with an off-the-shelf solver, yielding the scaling plan for the next interval.

Fernandez et al. [159] fuse threshold-based feedback on current performance with time series forecasts of demand inside a single Scaler that produces scale-in and scale-out decisions for heterogeneous Cloud infrastructures. An adaptive Predictor evaluates linear regression, ARMA, Holt–Winters, and autoregression on recent prediction errors and applies the best-performing variant to estimate medium-term demand. A Profiler characterises each VM type by its average CPU consumption per request, measured under non-saturated conditions (CPU utilisation below 75% and response time within 25–75% of the SLA target), and ranks VM types by processing capacity.

At every control step, the Scaler combines three signals to produce scaling plans: current utilisation and response time checked against thresholds, the predicted workload over a short planning horizon, and a decision-tree search over feasible combinations of VM types. Candidate plans are filtered by constraints on maximum acceptable utilisation and by billing-aware cost checks that prevent releasing recently launched resources, and are then ranked via a cost function trading monetary expense against expected utilisation, with the final choice keyed to a user-selected QoS class (bronze, silver, or gold). A Dynamic Load Balancer distributes incoming requests across the selected mix of VMs, with weights proportional to the capacities returned by the Profiler.

CHAPTER 4

PROBLEM DEFINITION AND METHODOLOGY

This chapter formalizes the research questions addressed in this thesis work, that is, how to schedule batch workloads of multiple workflows over a heterogeneous Cloud-Edge infrastructure to optimize execution time and cost, and how to dynamically provision Cloud resources for interactive workloads to satisfy performance requirements under fluctuating demand.

The chapter will first present the methodological approach proposed to solve the workflow scheduling problem. The proposed autoscaling strategy will be illustrated next.

4.1 Models for Workflow Scheduling

This section formalizes batch workloads, the heterogeneous Cloud-Edge resources on which they execute, and the execution time and monetary cost of the former on the latter under performance uncertainty. The models build on the concepts introduced in Section 2.1 and Section 2.2. Table 4.1 summarizes the notation used throughout this section.

4.1.1 WORKLOAD MODEL

The workload $\mathcal{L}$ consists of n independent workflows f_i ($i = 1, \dots, n$). Each workflow f_i is modeled as a DAG with k_i tasks T_j^i ($j = 1, \dots, k_i$). Figure 4.1 shows an example of a workload $\mathcal{L}$ composed of two workflows f_1 and f_2 composed of three and five tasks each.

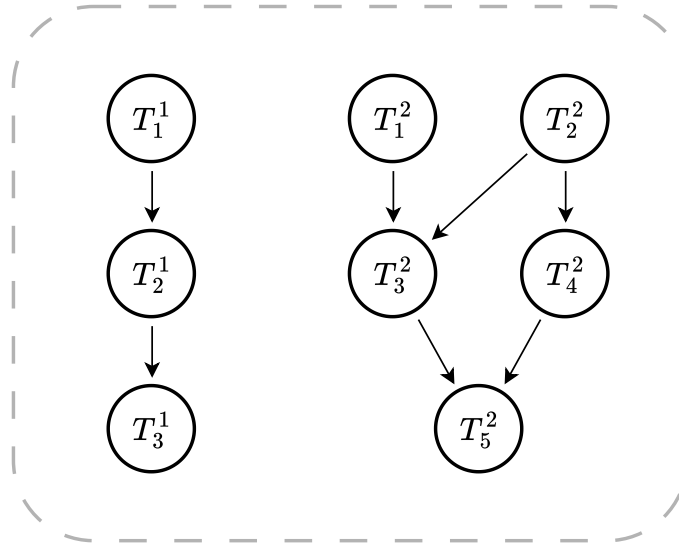


FIGURE 4.1: Example of a workload composed of two independent workflows, each represented as a DAG.

Each task T_j^i of workflow f_i is described by its demands:

$D_{i,j}^{\text{proc}}$: processing demand;

$D_{i,j,k}^{\text{xfer}}$: volume of data transferred to a task T_k^i ;

Workload model

$\mathcal{L}$	Workload
f_i	Workflow i
n	Number of workflows in $\mathcal{L}$
k_i	Number of tasks in workflow f_i
T_j^i	Task j of workflow f_i
$D_{i,j}^{\text{proc}}$	Processing demand of task T_j^i
$D_{i,j,k}^{\text{xfer}}$	Data volume transferred from task T_j^i to task T_k^i
$D_{i,j}^{\text{in}}$	Data volume transferred from external data sources to task T_j^i
$D_{i,j}^{\text{out}}$	Data volume transferred from task T_j^i to external storage devices

Infrastructure model

$\mathcal{R}$	Set of Cloud-Edge resources
r_l	Resource l in $\mathcal{R}$
h	Number of resources
V_l^{proc}	Processing capacity of resource r_l
$V_{l,m}^{\text{xfer}}$	Bandwidth between resources r_l and r_m
V_l^{in}	Bandwidth of resource r_l from external data sources
V_l^{out}	Bandwidth of resource r_l to external storage devices
c_l^{proc}	Leasing cost per unit time of resource r_l
$c_{l,m}^{\text{xfer}}$	Data transfer cost between resources r_l and r_m
c_l^{in}	Data transfer cost between r_l and external data sources
c_l^{out}	Data transfer cost between r_l and external storage devices

Execution model

$t_{i,j}^{\text{proc}}(r_l)$	Processing time of task T_j^i on resource r_l
$t_{i,j,k}^{\text{xfer}}(r_l, r_m)$	Data transfer time from task T_j^i on r_l to task T_k^i on r_m
$t_{i,j}^{\text{in}}(r_l)$	Data transfer time of task T_j^i from external data sources
$t_{i,j}^{\text{out}}(r_l)$	Data transfer time of task T_j^i to external storage devices
$t_j^i(s)$	Overall execution time of task T_j^i under scheduling plan s
$c_j^i(s)$	Overall monetary cost of task T_j^i under scheduling plan s
$\text{FT}(T_j^i, s)$	Finish time of task T_j^i under scheduling plan s
s	Scheduling plan
$\mathcal{T}_{\mathcal{L}}(s)$	Execution time (makespan) of workload $\mathcal{L}$ under scheduling plan s
$\mathcal{C}_{\mathcal{L}}(s)$	Overall monetary cost of workload $\mathcal{L}$ under scheduling plan s
$\mathcal{D}_{f_i}(s)$	Completion time of workflow f_i under scheduling plan s
d_{f_i}	Deadline of workflow f_i

TABLE 4.1: Summary of the main notation used in the context of workflow scheduling.

- $D_{i,j}^{\text{in}}$: volume of data received from external data sources;
- $D_{i,j}^{\text{out}}$: volume of data transferred to external storage devices.

4.1.2 CLOUD-EDGE MODEL

The Cloud-Edge infrastructure is modeled as a set $\mathcal{R}$ of computing resources r_l ($l = 1, \dots, h$) distributed across the Cloud and Fog/Edge layers of the Continuum.

Each resource r_l is described by the following parameters:

- V_l^{proc} : processing capacity;
- $V_{l,m}^{\text{xfer}}$: network bandwidth between resource r_l and r_m ;
- V_l^{in} : network bandwidth from external data sources;
- V_l^{out} : network bandwidth towards external storage devices.

Each resource r_l is also characterized by costs for leasing and data transfers:

- c_l^{proc} : leasing cost per unit time for using the resource;
- $c_{l,m}^{\text{xfer}}$: cost for transfers between resource r_l and resource r_m ;
- c_l^{in} : cost for retrieving data from external data sources;
- c_l^{out} : cost for transferring data to external storage devices.

Since resource performance is subject to variability, the parameters V_l^{proc} , $V_{l,m}^{\text{xfer}}$, V_l^{in} , and V_l^{out} are modeled as random variables.

4.1.3 EXECUTION AND COST MODELS

When a task T_j^i is mapped to a resource r_l , the time and monetary cost for its execution can be derived from the task demands and the resource characteristics.

The processing time of task T_j^i is given by the ratio between the task processing demand and the processing capacity of the resource, that is:

$$t_{i,j}^{\text{proc}}(r_l) = \frac{D_{i,j}^{\text{proc}}}{V_l^{\text{proc}}}$$

The data transfer times from external data sources and towards external storage devices are similarly obtained as ratios between the corresponding data volumes and the available bandwidths, that is:

$$t_{i,j}^{\text{in}}(r_l) = \frac{D_{i,j}^{\text{in}}}{V_l^{\text{in}}}, \quad t_{i,j}^{\text{out}}(r_l) = \frac{D_{i,j}^{\text{out}}}{V_l^{\text{out}}}$$

The data transfer time between two dependent tasks T_j^i and $T_k^i \in \text{succ}(T_j^i)$, mapped to resources r_l and r_m respectively is obtained as follows, taken to be zero when the two tasks share the same resource:

$$t_{i,j,k}^{\text{xfer}}(r_l, r_m) = \begin{cases} \frac{D_{i,j,k}^{\text{xfer}}}{V_{l,m}^{\text{xfer}}} & \text{if } r_l \neq r_m \\ 0 & \text{otherwise} \end{cases}$$

A scheduling plan s specifies the mappings between tasks and resources, and the task execution order.

The overall execution time of task T_j^i under scheduling plan s is then obtained as the sum of the processing time, the transfer times from external inputs and outputs, and the transfer times to all successor tasks:

$$t_j^i(s) = t_{i,j}^{\text{proc}}(r_l) + \sum_{T_k^i \in \text{succ}(T_j^i)} t_{i,j,k}^{\text{xfer}}(r_l, r_m) + t_{i,j}^{\text{in}}(r_l) + t_{i,j}^{\text{out}}(r_l)$$

The monetary cost $c_j^i(s)$ of executing task T_j^i under scheduling plan s accounts for two components: the leasing cost of the resource, and the data transfer costs, that is:

$$c_j^i(s) = c_l^{\text{proc}} t_j^i(s) + \sum_{T_k^i \in \text{succ}(T_j^i)} c_{l,m}^{\text{xfer}} D_{i,j,k}^{\text{xfer}} + c_l^{\text{in}} D_{i,j}^{\text{in}} + c_l^{\text{out}} D_{i,j}^{\text{out}}$$

Since the resource performance parameters are random variables, the execution times and monetary costs derived are themselves random variables.

The completion time of workflow f_i under scheduling plan s is defined as the maximum finish time of its tasks, that is:

$$\mathcal{D}_{f_i}(s) = \max_{j \in \{1, \dots, k_i\}} \text{FT}(T_j^i, s)$$

where $\text{FT}(T_j^i, s)$ denotes the finish time of task T_j^i . The finish time $\text{FT}(T_j^i, s)$ of each task depends on its start time and how long it takes to execute. Each task starts at its earliest possible time, that is, as soon as both its assigned resource r_l is available and all its predecessors have completed their execution.

Task execution is non-preemptive and each resource processes one task at a time. The scheduling order itself must be a linear ordering of all tasks in $\mathcal{L}$ that preserves the topological order of each workflow, that is, for every edge (T_j^i, T_k^i) , task T_j^i appears before task T_k^i .

The makespan $\mathcal{T}_{\mathcal{L}}(s)$ of the entire workload $\mathcal{L}$ is then given by the maximum completion time across all workflows:

$$\mathcal{T}_{\mathcal{L}}(s) = \max_{i \in \{1, \dots, n\}} \mathcal{D}_{f_i}(s)$$

Since tasks from different workflows may share resources, both $\mathcal{D}_{f_i}(s)$ and $\mathcal{T}_{\mathcal{L}}(s)$ depend on the entire scheduling plan, not only on the tasks belonging to a single workflow.

The overall monetary cost of the execution of workload $\mathcal{L}$ under the scheduling plan s is the sum of the costs of all tasks:

$$\mathcal{C}_{\mathcal{L}}(s) = \sum_{i=1}^n \sum_{j=1}^{k_i} c_j^i(s)$$

As task execution times and costs are random variables, both $\mathcal{T}_{\mathcal{L}}(s)$ and $\mathcal{C}_{\mathcal{L}}(s)$ are random variables.

4.2 Multi-Objective Workflow Scheduling Problem

The models previously defined provide the foundation to formulate the optimization problem whose goal is to identify scheduling plans that offer the best tradeoffs between two conflicting objectives: workload execution time and monetary cost of the Cloud-Edge resources.

Figure 4.2 provides an overview of the proposed scheduling framework. As can be seen, the solution of the scheduling problem, formulated as a multi-objective optimization problem, is based on the application of a Genetic Algorithm (GA) combined with a Monte Carlo simulation. The Pareto front of non-dominated scheduling plans represents the best tradeoffs identified by the GA.

4.2.1 PROBLEM FORMULATION

The multi-objective optimization problem is formulated as follows:

$$\begin{aligned} & \text{minimize} && (P_p[\mathcal{T}_{\mathcal{L}}(s)], P_q[\mathcal{C}_{\mathcal{L}}(s)]) \\ & \text{subject to} && \mathbb{E}[\mathcal{D}_{f_i}(s)] \leq d_{f_i}, \quad \forall f_i \in \mathcal{L}_d \end{aligned}$$

where P_p and P_q denote the p -th and q -th percentiles of the probability distributions of the workload execution time and monetary cost, respectively, and $\mathcal{L}_d \subseteq \mathcal{L}$ is the subset of workflows that carry a deadline d_{f_i} . The subset $\mathcal{L}_d$ may include any number of workflows, from none to all of those in $\mathcal{L}$; workflows outside $\mathcal{L}_d$ are unconstrained and contribute to the formulation only through the workload-level objectives.

Targeting percentiles rather than expected values allows users to select the probability with which execution time and cost will not exceed the optimized values, thereby controlling how conservatively the scheduling plan accounts for performance variability.

Since the two objectives are minimized simultaneously, candidate scheduling plans are compared through Pareto dominance [35]: given two feasible scheduling plans s_1 and s_2 , s_1 is said to dominate s_2 if s_1 performs at least as well as s_2 on both objectives and strictly better on at least one. Formally:

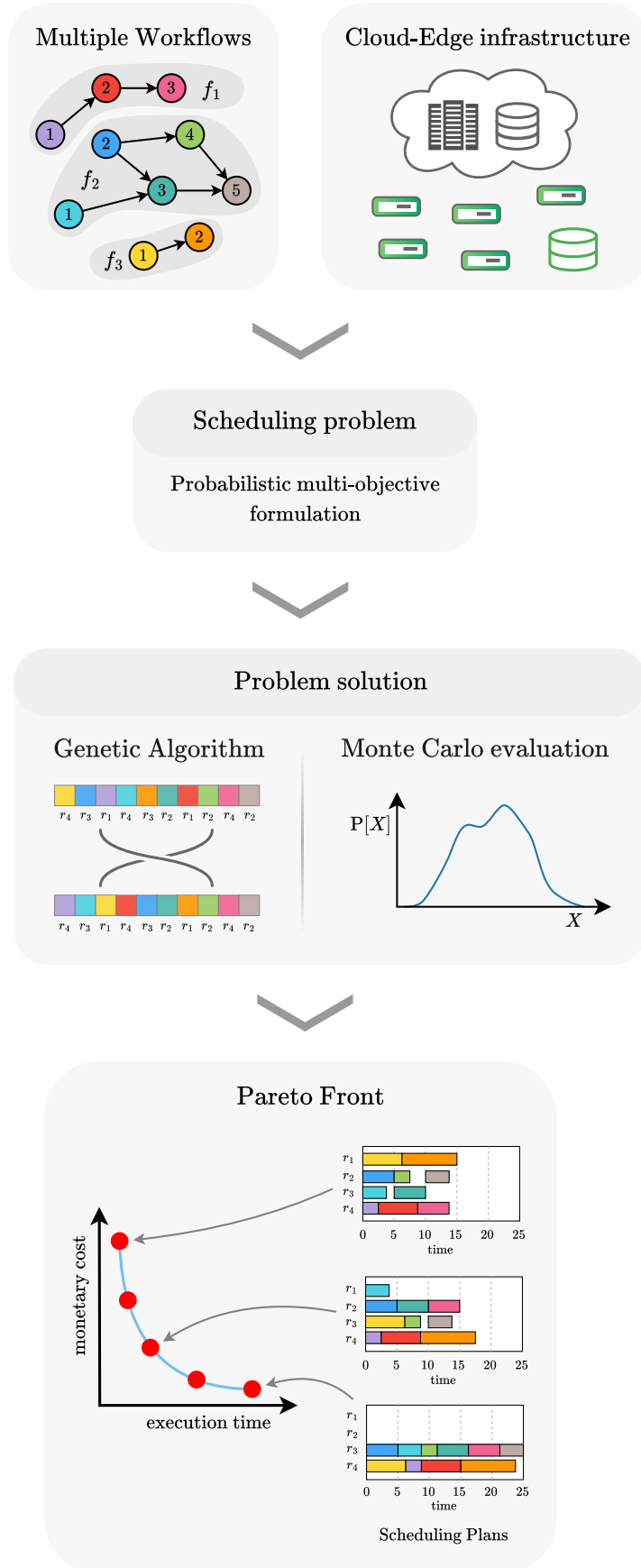


FIGURE 4.2: Overview of the proposed scheduling framework.

$$s_1 \prec s_2 \iff \begin{cases} P_p[\mathcal{T}_{\mathcal{L}}(s_1)] \leq P_p[\mathcal{T}_{\mathcal{L}}(s_2)] \\ P_q[\mathcal{C}_{\mathcal{L}}(s_1)] \leq P_q[\mathcal{C}_{\mathcal{L}}(s_2)] \end{cases}$$

with at least one strict inequality.

A feasible scheduling plan that is not dominated by any other is called Pareto optimal (or non-dominated). The set of all Pareto optimal scheduling plans constitutes the Pareto front.

4.2.2 PROBLEM STRUCTURE

The solution space of the optimization problem consists of all valid combinations of task scheduling orders and task-to-resource mappings. Formally, the solution space is defined as:

$$\mathcal{S} = S_K \times [1, \dots, h]^K$$

where $K = \sum_{i=1}^n k_i$ is the total number of tasks across all workflows, S_K is the set of valid scheduling orders, and $[1, \dots, h]^K$ represents all possible mappings of K tasks to h resources.

A valid task scheduling order is any permutation of the K tasks that constitutes a topological sort of the dependency graph.

The number of valid topological sorts grows combinatorially with the number of tasks and the structure of the dependency graphs, and the resource mapping space grows exponentially with both the number of tasks and the number of resources. Together, these factors cause the solution space to grow by several orders of magnitude even for modest increases in the number of tasks or resources.

The optimization problem presents two main difficulties. First, workflow scheduling is NP-hard (see Section 2.3), and the multi-workflow setting further enlarges the search space. Second, the probabilistic nature of the objectives requires the estimation of probability distributions for each candidate scheduling plan. The exact computation of these distributions is generally intractable due to the dependencies among task execution times, which involve maxima over predecessor completion times and sums of execution and

transfer durations.

Heuristic or meta-heuristic approaches are therefore employed to explore the search space, while simulation-based methods are used to estimate the required distributions empirically.

4.2.3 SOLUTION APPROACH

The combinatorial nature of the solution space and the intractability of the probabilistic objectives call for a population-based meta-heuristic coupled with a simulation-based evaluation procedure.

A Genetic Algorithm (GA) is adopted as the meta-heuristic. Since the probability distributions of $\mathcal{T}_{\mathcal{L}}(s)$ and $\mathcal{C}_{\mathcal{L}}(s)$ cannot be derived analytically, the fitness of each candidate scheduling plan is evaluated through a Monte Carlo simulation.

4.3 Autoscaling Strategy

The autoscaling problem is defined for a Cloud computing environment in which a service handles incoming requests using a set of resources whose number is adjusted over time. Table 4.2 summarizes the notation used in the remainder of the chapter.

4.3.1 WORKLOAD AND INFRASTRUCTURE MODELS

The workload is represented as a continuous stream of incoming requests whose arrival rate varies over time. The autoscaler is invoked at regular instants separated by an interval δ , which partitions time into time slots of equal duration. In time slot i starting at time t_i , the workload intensity is characterized by the arrival rate λ_i , defined as the average number of requests per unit time and assumed constant within the slot.

Each request has a service demand S , i.e., the average time required to process one request on a single resource.

Within each time slot i , the autoscaler controls the number of allocated resources R_i . Each resource has a processing capacity μ , i.e., the maximum number of requests it can serve per unit time, so that $S = 1/\mu$; the allocated

δ	Time slot duration
t_i	Start time of slot i
h	Time horizon
λ_i	Arrival rate in slot i
S	Request service demand
μ	Processing capacity of a resource (maximum requests per unit time)
R_i	Number of allocated resources in slot i
X_i	Aggregate throughput in slot i (completed requests per unit time)
U_i	Resource utilization in slot i
U^{ref}	Desired resource utilization
Q_i	Backlog of unprocessed requests in slot i
$\hat{\lambda}_j$	Predicted arrival rate in slot j
$\hat{Q}_j$	Predicted backlog in slot j
$\hat{N}_j$	Predicted number of unprocessed requests in slot j
$\hat{X}_{i+h}$	Predicted throughput at the prediction horizon
$\hat{U}_{i+h}$	Predicted resource utilization at the prediction horizon
$\hat{R}_{i+h}$	Predicted number of required resources at the prediction horizon
K	Number of VM types (heterogeneous scenario)
R_i^k	Number of allocated resources of type k in slot i
μ_k	Processing capacity of VM type k
p_k	Leasing cost of VM type k per unit time
S_k	Service demand on VM type k
$\hat{X}_{i+h}^*$	Capacity target at the prediction horizon

TABLE 4.2: Summary of the main notation used in the context of autoscaling.

resources at time slot i therefore provide an aggregate processing capacity of $R_i \mu$. When resources are added, they require h time slots of duration δ before becoming operational.

Together with the arrival rate, S links workload intensity to resource utilization as formalized in the following.

4.3.2 CAPACITY MANAGEMENT

The autoscaler operates on the resource utilization U , regulating it around a desired level U^{ref} , with $0 < U^{\text{ref}} < 1$. According to the utilization law [160],

U equals the product of the throughput X , i.e., the rate at which the system processes incoming requests, and the service demand S . Hence, in the time slot i starting at time t_i , the utilization is given by:

$$U_i = X_i S$$

Evaluating U_i thus reduces to estimating the throughput X_i . In steady-state conditions, the throughput coincides with the arrival rate λ_i . However, steady-state conditions cannot always be assumed: during load spikes or while newly added resources are still booting, the allocated resources may be unable to serve all incoming requests within slot i . The unserved requests accumulate into a backlog Q_i that carries over to subsequent slots, and the throughput then depends on both the current arrival rate and the previously accumulated backlog.

A capacity estimate based on λ_i alone therefore underestimates the actual demand; the framework addresses this by propagating the backlog step by step into a capacity estimation procedure.

4.4 Load-Aware Predictive Autoscaling Strategy

The proposed Load-Aware Predictive framework implements a control loop that anticipates the future arrival rate and adjusts the allocated resources accordingly. The framework aligns its prediction horizon with the VM boot time, set equal to h time slots, so that scaling decisions are issued early enough for newly provisioned resources to become available precisely when the predicted load materializes.

Figure 4.3 illustrates the framework. At every time slot of duration δ , a workload predictor consumes the most recent arrival rate measurements and forecasts the arrival rate h time slots ahead. The forecast, together with the measured backlog of unprocessed requests, enters a capacity estimator that computes the number of resources required to maintain the desired utilization level U^{ref} . Incorporating the backlog ensures that the capacity estimate accounts for accumulated unprocessed requests rather than relying solely on the predicted arrival rate. A resource selector then compares this requirement against the

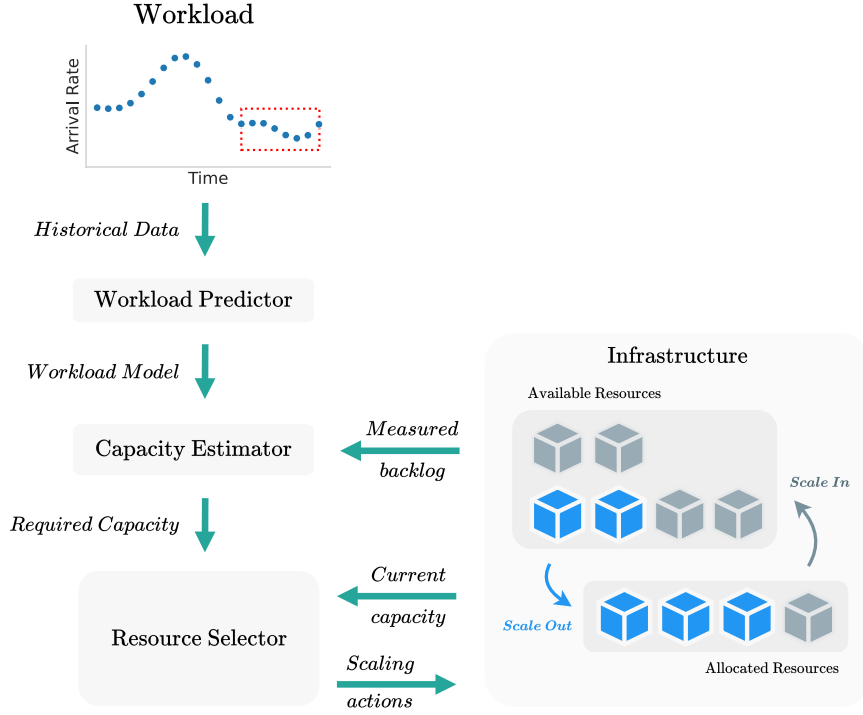


FIGURE 4.3: Overview of the proposed autoscaling framework.

number of allocated resources and determines the appropriate scaling action: whether to scale-out, scale-in, or maintain the current allocation.

In detail, starting from the current time slot i , the workload predictor estimates the arrival rate $\hat{\lambda}_j$ for each slot $j = i + 1, \dots, i + h$ within the prediction horizon, using a model fitted to a rolling window of the most recent arrival rate observations to allow forecasts to track evolving workload patterns.

The capacity estimator derives the number of resources required to satisfy the desired utilization level U^{ref} by computing the number of unprocessed requests at each future time slot $j = i + 1, \dots, i + h$ within the prediction horizon, defined as the sum of the predicted backlog and the predicted new arrivals:

$$\hat{N}_j = \hat{Q}_{j-1} + \hat{\lambda}_j \delta$$

In particular, while the backlog at the current time slot i is measured, the backlog at future time slots is predicted by iteratively propagating the current back-

log through each slot, accounting for the throughput of the allocated resources at each slot. Once the request backlog and the number of unprocessed requests are estimated at the horizon, the predicted throughput at the horizon is given by:

$$\hat{X}_{i+h} = \frac{\hat{N}_{i+h}}{\delta}$$

Applying the utilization law yields the predicted utilization $\hat{U}_{i+h} = \hat{X}_{i+h} S$. The number of resources required is then:

$$\hat{R}_{i+h} = \left\lceil \frac{\hat{U}_{i+h}}{U_{\text{ref}}} \right\rceil$$

where the ceiling function ensures a discrete resource count.

By comparing $\hat{R}_{i+h}$ with the number of currently allocated resources, the resource selector implements the strategy determining whether and to what extent a scaling action is needed. The scaling policies are "asymmetric": scale-out actions are triggered promptly whenever the predicted capacity exceeds that of the available resources, whereas scale-in actions are conservative, requiring persistent over-provisioning across multiple future time slots before any resource is deallocated. Additional operational constraints, such as cool-down intervals and minimum allocation times, are defined to prevent oscillatory behavior.

4.5 Autoscaling with Heterogeneous Resources

Cloud providers offer catalogs of VM types that differ in processing capacity and leasing cost. Exposing this diversity to the autoscaler enables finer-grained tracking of workload fluctuations, since the same aggregate capacity can be assembled from alternative mixes of resources at different costs, allowing the autoscaler to favor combinations that match the predicted demand at the lowest cost.

In the heterogeneous resource scenario, the resource allocation at time slot i can span K VM types and is described by the per-type counts R_i^k , $k = 1, \dots, K$. Resources of type k have processing capacity μ_k and leasing cost p_k per unit time, with per-type service demand $S_k = 1/\mu_k$. The boot time h is shared across types. The aggregate processing capacity in time slot i is $\sum_{k=1}^K R_i^k \mu_k$.

Equal aggregate capacity can be produced by different combinations of R_i^k , so the natural target of the load-aware estimation is the capacity itself rather than a resource count. Consequently, a capacity target is derived from the predicted number of unprocessed requests $\hat{N}_{i+h}$ at the horizon, the time slot duration δ , and the desired utilization level U^{ref} :

$$\hat{X}_{i+h}^* = \frac{\hat{N}_{i+h}}{\delta U^{\text{ref}}}$$

The capacity target $\hat{X}_{i+h}^*$ decouples the load-aware estimation from the discrete provisioning decision; its translation into a concrete mix of VM types is delegated to the resource selector, which determines the VM mix that bridges the gap between $\hat{X}_{i+h}^*$ and the current aggregate capacity.

For scale-out actions, the mix with the smallest aggregate capacity sufficient to cover the gap at the lowest leasing cost is selected. For scale-in actions, resources are deallocated starting from the most powerful and expensive types until the desired capacity reduction is reached. The detailed selection procedure is presented in Chapter 7.

CHAPTER 5

WORKFLOW SCHEDULING FRAMEWORK

As already stated, the solution approach proposed to solve the scheduling problem combines a GA meta-heuristic for exploring the combinatorial solution space with the Monte Carlo simulation for estimating the probabilistic objectives. This chapter presents the detailed design of the solution framework.

5.1 Scheduling Plan Encoding

A fundamental step of GAs is the encoding of the population individuals, that is, how a solution of the optimization problem is represented in the form of a chromosome. In the context of workflow scheduling, each individual (i.e., a candidate scheduling plan) consists of two distinct components: a scheduling chromosome that encodes the task scheduling order, and a provisioning chromosome that encodes the task-to-resource mapping.

Figure 5.1 illustrates this for a workflow composed of six tasks: the DAG appears in (a) and the chromosome encoding of a possible scheduling plan in (b).

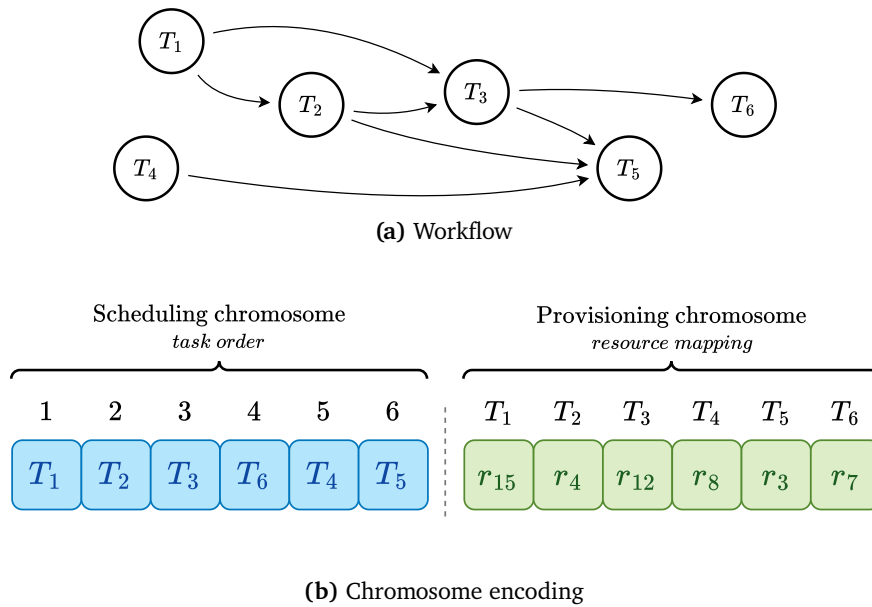


FIGURE 5.1: Chromosome encoding of a scheduling plan for an example workflow.

The index in the scheduling chromosome denotes the task scheduling order, while the provisioning chromosome maps each task to the allocated resource.

In the example, task T_1 is scheduled first on resource r_{15} , followed by T_2 on r_4 , T_3 on r_{12} , T_6 on r_7 , T_4 on r_8 , and T_5 last on r_3 .

For workloads composed of multiple workflows, the scheduling chromosome might interleave tasks from distinct workflows while preserving the internal dependencies of each workflow.

5.2 Evolutionary Process

The GA evolves a fixed-size population of scheduling plans over successive generations. Each generation cycles through parent selection, which identifies the individuals from the current population that will be used to produce offspring; offspring production by crossover and mutation; and survival selection, which determines which individuals from the combined parent and offspring populations are carried over to the next generation. The initial population is evaluated in advance, and each generation evaluates only its new offspring, so selection always operates on fully evaluated individuals. Both parent and survival selection phases follow the NSGA-II scheme [35], comparing individuals by Pareto rank and, at equal rank, by crowding distance to preserve diversity.

The parent selection follows the binary tournament mechanism: two individuals are drawn at random, compared by ranking and crowding distance, and the winner becomes a parent for crossover.

Crossover and mutation operators are applied independently to the two chromosomes of each individual due to their different structural properties: the provisioning chromosome admits any combination of resource assignments, whereas the scheduling chromosome must always represent a valid topological sort of the workflow DAGs. Standard operators are applied to the provisioning chromosome, while the scheduling chromosome requires customized operators.

5.2.1 Crossover Operators

Figure 5.2 illustrates the results of the application of the two crossover operators to a pair of parents P_1 and P_2 , producing offspring O_1 and O_2 at crossover point $\kappa = 3$.

Crossover on the scheduling chromosome is implemented as a customized operator that guarantees the offspring ordering remains a valid topological sort [27]. Given a random crossover point κ , the first offspring O_1 inherits the first κ tasks from parent P_1 while the remaining positions are filled by appending the tasks not yet scheduled, so as to preserve their relative order in P_2 . As a consequence, the resulting offspring satisfy DAG dependencies.

For the provisioning chromosome, standard single-point crossover is applied.

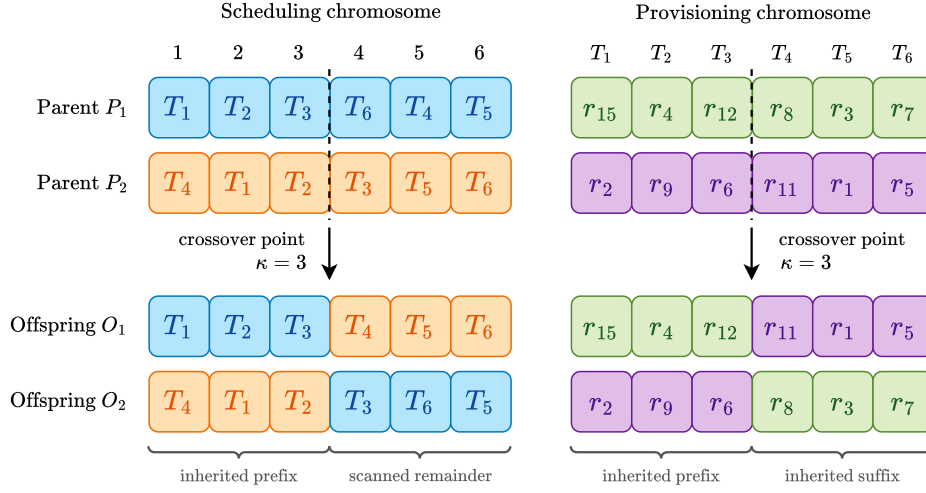


FIGURE 5.2: Crossover operators applied to scheduling and provisioning chromosomes.

5.2.2 MUTATION OPERATORS

Figure 5.3 illustrates the two mutation operators applied to the offspring O_1 shown in Figure 5.2.

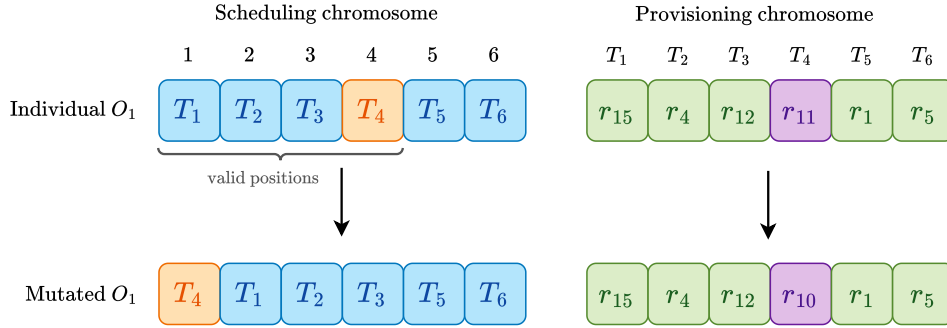


FIGURE 5.3: Mutation operators applied to scheduling and provisioning chromosomes.

For the scheduling chromosome, a task at a random position is selected. The order is analyzed in both directions from this position, stopping on each side at the nearest predecessor or successor of the selected task. This defines a contiguous range of valid positions whose tasks are neither predecessors nor successors. The task is then moved to a position sampled uniformly within this range. Because the range is delimited by the dependencies of the selected

task on both sides, the resulting order remains a valid topological sort. In the example, the selected task is T_4 at position 4, which does not have any predecessor. Its nearest successor T_5 is at position 5, thus the valid positions span 1 to 4. T_4 is moved to position 1.

For the provisioning chromosome, each task is mutated independently: the allocated resource is replaced by a resource uniformly sampled from the set of available resources. In the example, r_{11} is replaced by r_{10} .

5.3 Scheduling Plan Evaluation

Each candidate scheduling plan s is evaluated on two objectives, i.e., the p -th percentile of the workload execution time $\mathcal{T}_{\mathcal{L}}(s)$ and the q -th percentile of the monetary cost $\mathcal{C}_{\mathcal{L}}(s)$ (see Section 4.2), used in the fitness vector:

$$\text{fitness}(s) = (P_j[\mathcal{T}_{\mathcal{L}}(s)], P_l[\mathcal{C}_{\mathcal{L}}(s)])$$

Since the fitness components are percentiles of random variables whose distributions cannot be derived in closed form, they are estimated empirically via Monte Carlo simulation. These estimates are obtained by drawing repeated samples of the input distributions and aggregating the resulting outputs into an empirical distribution.

In detail, given a scheduling plan s , the expressions for $\mathcal{T}_{\mathcal{L}}(s)$ and $\mathcal{C}_{\mathcal{L}}(s)$ are derived as functions of the random variables that model the resource performance. These random variables are sampled according to the specified probability distributions, and each sample is substituted into the corresponding expressions to derive one realization of the workload execution time and monetary cost. The percentiles P_j and P_l are extracted from the resulting empirical distributions to form the fitness vector.

As an example, these expressions are derived for the workload in Figure 4.1, whose workflows f_1 and f_2 consist of three and five tasks, under a candidate scheduling plan s that maps them onto four resources $r_1, \dots, r_4$ with the following task scheduling order:

$$[T_1^2, T_1^1, T_2^2, T_2^1, T_3^2, T_4^2, T_3^1, T_5^2]$$

and the following task-to-resource mappings:

$$\begin{aligned} \{T_1^1, T_3^1\} &\rightarrow r_1 & \{T_2^2, T_4^2, T_5^2\} &\rightarrow r_2 \\ \{T_1^2, T_3^2\} &\rightarrow r_3 & \{T_2^1\} &\rightarrow r_4 \end{aligned}$$

The workload execution time is given by:

$$\mathcal{T}_{\mathcal{L}}(s) = \max\{\mathcal{D}_{f_1}(s), \mathcal{D}_{f_2}(s)\}$$

The completion time of each workflow is determined by the finish times of its constituent tasks, taking into account both dependencies and resource contention. For workflow f_1 , whose three tasks are allocated to resources r_1 and r_4 , the completion time is the sum of the task execution times since they must be executed sequentially, that is:

$$\mathcal{D}_{f_1}(s) = t_1^1(r_1) + t_2^1(r_4) + t_3^1(r_1)$$

For workflow f_2 , the expression of its completion time is given by:

$$\mathcal{D}_{f_2}(s) = \max\{\max\{t_1^2(r_3), t_2^2(r_2)\} + t_3^2(r_3), t_2^2(r_2) + t_4^2(r_2)\} + t_5^2(r_2)$$

The inner maximum accounts for the fact that task T_3^2 can begin only after the later of its two predecessors T_1^2 and T_2^2 has completed, while the outer maximum reflects that T_5^2 can start only after both T_3^2 and T_4^2 have completed their execution.

The distribution of $\mathcal{C}_{\mathcal{L}}(s)$ is estimated in a similar way from the same Monte Carlo samples.

5.4 Initial Population

The selection of the initial population plays a critical role in the convergence of the GA. A population that is too homogeneous may cause the algorithm to limit the search in a narrow region of the solution space, while a diversified one allows the GA to explore larger regions.

The generation of the initial population combines two complementary mechanisms: a scheduling order generator, which arranges the tasks into valid topological sorts, and a resource provisioning generator, which determines task-to-resource mappings.

The scheduling order generator is based on Algorithm 1. The algorithm main-

Algorithm 1 Topological sort generator

Require: $\mathcal{L}$: workload

Ensure: $taskSort$: a topological sort of tasks in $\mathcal{L}$

```

1:  $readyTasks \leftarrow$  all entry tasks in  $\mathcal{L}$ 
2:  $taskSort \leftarrow \{\}$ 
3: while  $readyTasks$  is not empty do
4:    $t \leftarrow \text{EXTRACTTASK}(readyTasks)$ 
5:   append  $t$  to  $taskSort$ 
6:   for  $neighbor \in \text{successors}(t, \mathcal{L})$  do
7:     if  $\text{predecessors}(neighbor, \mathcal{L}) \subseteq taskSort$  then
8:       add  $neighbor$  to  $readyTasks$ 
9:     end if
10:  end for
11: end while
12: return  $taskSort$ 

```

tains a set of ready tasks, initialized with the entry tasks of every workflow in $\mathcal{L}$. At each iteration of the main loop, a task is extracted from this set by the `EXTRACTTASK` subroutine and appended to $taskSort$. Every successor of the extracted task whose predecessors are already in $taskSort$ is then added to the ready set, making it eligible for extraction in subsequent iterations. The loop terminates once the ready set is empty, that is, all tasks of the workflow have been allocated.

Two selection policies are considered for the extraction step. The first policy, *random selection from different workflows*, is detailed in Algorithm 2. The policy uniformly samples one workflow among those with at least one ready task, and then uniformly samples one of the ready tasks of that workflow. The resulting task scheduling order interleaves tasks of different workflows.

The second policy, *random selection from preferential workflows*, assumes priorities assigned to the workflows. The tasks of the highest-priority workflow are allocated first.

The resource provisioning generator complements the scheduling order generator by including in the initial population scheduling plans that bias the search

Algorithm 2 Random selection from different workflows**Require:** *readyTasks*: set of tasks ready to be scheduled**Ensure:** *t*: selected task

```

1: function EXTRACTTASK(readyTasks)
2:    $w \leftarrow$  list of workflows in readyTasks
3:    $w_i \leftarrow$  UNIFORMSAMPLING( $w$ )
4:   tasks  $\leftarrow$  tasks of workflow  $w_i$  in readyTasks
5:    $t \leftarrow$  UNIFORMSAMPLING(tasks)
6:   return t
7: end function

```

toward the extremes of the objective space. In addition to randomly generated provisioning chromosomes, four deterministic strategies are included:

- Sequential execution on the best-performing resource: all tasks are assigned to the resource with the highest processing capacity.
- Fully parallel execution on best-performing resources: each task is assigned to a resource of the highest-capacity type.
- Sequential execution on the cheapest resource: all tasks are assigned to the resource with the lowest leasing cost.
- Fully parallel execution on cheapest resources: each task is assigned to a different resource with the lowest leasing cost.

CHAPTER 6

EXPERIMENTAL EVALUATION OF WORKFLOW SCHEDULING

This chapter describes the evaluation of the proposed workflow scheduling strategy on synthetic and real scientific workloads, composed of either single or multiple workflows, under both deterministic and probabilistic performance assumptions. The evaluation quantifies the tradeoff between workload execution time and monetary cost obtained when scheduling these workloads across a heterogeneous Cloud-Edge infrastructure.

Two flavors of this workflow are considered, differing in their Communication-to-Computation Ratio, defined as:

$$\text{CCR}_f = \frac{\sum_j \overline{t_j^{\text{xfer}}}}{\sum_j \overline{t_j^{\text{proc}}}}$$

where $\overline{t_j^{\text{proc}}}$ is the estimated time to execute task T_j averaged over all resources of the infrastructure, and $\overline{t_j^{\text{xfer}}}$ is the estimated time to transfer the output data of T_j to its dependent tasks averaged over all pairs of resources. CCR_f thus relates the time a workflow spends on data transfer to the time it spends on processing across the infrastructure. A balanced flavor has $\text{CCR}_f = 1$, while a computation-intensive flavor has $\text{CCR}_f = 0.1$.

The experiments with multiple workflows use synthetic workloads derived from the DAG shown in Figure 6.2, comprising 26 tasks connected by 38 directed edges representing precedence constraints, with ten tasks receiving input data from external sources and three producing output data for external storage devices. Tasks exhibit heterogeneous processing demands and data volumes.

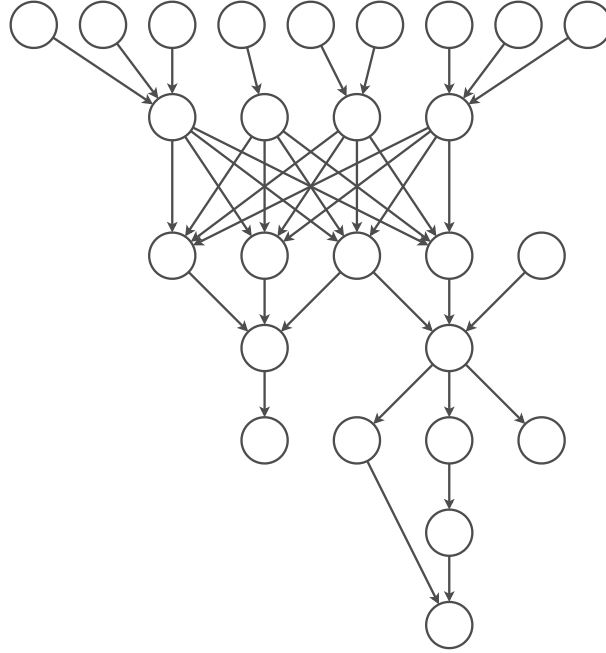


FIGURE 6.2: Structure of the artificially generated workflow used to derive the synthetic workloads.

Starting from this structure, three workflow flavors are defined, differing only in the magnitude of their task demands. The "lightest" flavor, f_1 , has processing demands ranging from 0.4 to 100 MI per task, with an average of 12.8 MI. In terms of data volumes, each task transfers on average 220 MBytes to its dependent tasks, while the aggregate input data retrieved from external data sources amounts to 9.4 GBytes and the aggregate output data transferred to external storage devices amounts to 4 GBytes.

The demands of the second flavor, f_2 , are 50% larger than those of f_1 , and those of the third flavor, f_3 , are twice as large. The three flavors thus share an identical DAG topology but exhibit progressively higher resource requirements.

Two sets of synthetic workloads are derived from these flavors. The first set consists of 12 workloads, each composed of one to four instances of a single flavor. Specifically, $\mathcal{L}_{f_1}$, $\mathcal{L}_{2f_1}$, $\mathcal{L}_{3f_1}$, and $\mathcal{L}_{4f_1}$ denote workloads containing one, two, three, and four instances of workflow f_1 , respectively. Analogous workloads are constructed for f_2 and f_3 , yielding 12 workloads in total. The number of tasks in each workload ranges from 26 (single-workflow) to 104 (four workflows).

The second set consists of four workloads composed of two or three distinct flavors: $\mathcal{L}_{f_1f_2}$, $\mathcal{L}_{f_1f_3}$, $\mathcal{L}_{f_2f_3}$, and $\mathcal{L}_{f_1f_2f_3}$. For instance, $\mathcal{L}_{f_1f_2}$ contains one instance of f_1 and one instance of f_2 , while $\mathcal{L}_{f_1f_2f_3}$ contains one instance of each of the three flavors. The number of tasks is either 52 or 78. Table 6.1 summarizes the aggregate processing demands and data volumes of these mixed workloads.

Workload	Processing	Data Volume		
	Demand [MI]	Task-Task [GBytes]	Input [GBytes]	Output [GBytes]
$\mathcal{L}_{f_1f_2}$	826.95	20.9	23.4	10.1
$\mathcal{L}_{f_1f_3}$	992.34	25.1	28.1	12.1
$\mathcal{L}_{f_2f_3}$	1,157.73	29.3	32.8	14.1
$\mathcal{L}_{f_1f_2f_3}$	1,488.51	37.6	42.1	18.1

TABLE 6.1: Overall processing demands and data volumes of the four mixed-workflow synthetic workloads.

REAL SCIENTIFIC WORKLOADS

In addition to the synthetic workloads, two real scientific workloads drawn from distinct application domains are considered, namely Cycles, from smart agriculture, and CyberShake, from seismology.

Cycles [161] is a high-throughput workflow used for agroecosystem simulation. The Cycles workflow, illustrated in Figure 6.3, models biophysical processes and agricultural management practices. Grey, green, and yellow tasks perform the core model simulations, whereas the remaining tasks summarize the model outputs and generate the corresponding diagrams.

Three flavors of the Cycles workflow are considered, denoted f_{A_1} , f_{A_2} , and

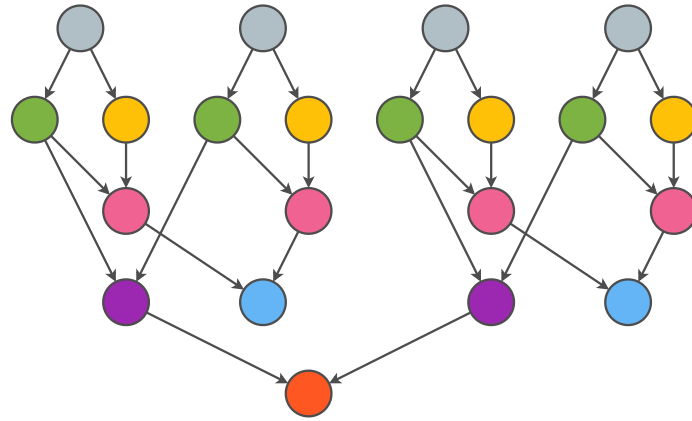


FIGURE 6.3: Structure of a representative Cycles workflow, with colors differentiating task activities.

f_{A_3} , consisting of 67, 133, and 331 tasks, respectively. Table 6.2 reports the aggregate processing demands and data volumes for each flavor, with task demands and data volumes being obtained from the data published in the *WfInstances* repository¹.

The workload $\mathcal{L}_A$ used in the experiments consists of the three flavors combined, for a total of 531 tasks.

The second scientific workload is based on CyberShake [162], a workflow that characterizes earthquake hazards through probabilistic seismic hazard analysis, illustrated in Figure 6.4. In CyberShake, tasks encompass the computation of seismogram syntheses, the collection and compression of the resulting syn-

¹<https://github.com/wfcommons/WfInstances/blob/main/pegasus/cycles/>

Workflow	Processing Demand [MI]	Data Volume		
		Task-Task [MBytes]	Input [MBytes]	Output [MBytes]
f_{A_1}	13.37	1.7	57.3	467.2
f_{A_2}	57.07	3.4	114.5	929.3
f_{A_3}	163.78	8.6	286.4	2,316.0

TABLE 6.2: Overall processing demands and data volumes of the three Cycles workflows.

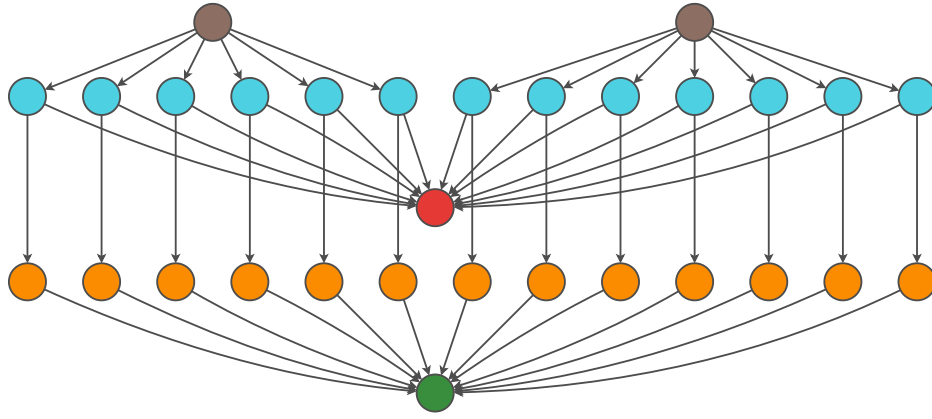


FIGURE 6.4: Structure of a representative CyberShake workflow, with colors differentiating task activities.

thetic seismograms, and subsequent processing.

Three flavors are considered, denoted f_{S_1} , f_{S_2} , and f_{S_3} , consisting of 30, 50, and 100 tasks, respectively. Table 6.3 reports their aggregate processing demands and data volumes, derived from the data published in the Pegasus Workflow Repository².

The workload $\mathcal{L}_S$ used in the experiments combines the three flavors, for a total of 180 tasks.

The two scientific workloads differ substantially in their characteristics. The Cycles workflows are dominated by external data transfers, with output volumes in the hundreds of megabytes to gigabytes range, while inter-task data volumes remain comparatively small. In contrast, the CyberShake workflows

²https://pegasus.isi.edu/workflow_gallery/

Workflow	Processing	Data Volume		
	Demand [MI]	Task-Task [GBytes]	Input [GBytes]	Output [GBytes]
f_{S_1}	10.32	6.2	80.2	31×10^{-6}
f_{S_2}	15.68	12.7	81.4	88×10^{-6}
f_{S_3}	33.69	22.8	80.8	161×10^{-6}

TABLE 6.3: Overall processing demands and data volumes of the three CyberShake workflows.

exhibit very large inter-task data volumes and input data volumes (i.e., tens of gigabytes), but negligible output volumes.

6.1.2 CLOUD-EDGE INFRASTRUCTURE

The infrastructure considered in the experiments follows the Cloud-Edge model defined in Section 4.1 and consists of six resource types, four in the Cloud layer and two in the Edge layer. Table 6.4 summarizes the processing capacity, bandwidth, and leasing cost of each resource type.

Resource Type	Processing Capacity [MIPS $\times 10^3$]	Bandwidth [Gbps]	Leasing Cost [USD/hr]
Cloud-4XLarge	400	20	4.20
Cloud-2XLarge	200	10	2.00
Cloud-Large	100	5	0.90
Cloud-Medium	50	5	0.40
Edge-Large	75	0.4	0.05
Edge-Small	10	0.2	0.005

TABLE 6.4: Characteristics of the Cloud-Edge resources considered in the experiments.

The processing capacity of each resource is expressed in Million Instructions Per Second (MIPS). Under this convention, the ratio between processing demand and processing capacity is derived as processing time in seconds (see Section 4.1).

Resource usage follows a discrete billing model commonly used by Cloud

providers [30] in which charges are rounded up to the nearest billing interval of duration u_l . The billable duration of task T_j^i on resource r_l is thus given by $u_l \lceil t_j^i(r_l)/u_l \rceil$, bounded below by u_l regardless of how short the actual execution time may be. In the experiments, $u_l = 60$ seconds (per-minute billing) for all resource types. The leasing cost term of the general cost model in Section 4.1 is thus specialized as $c_l^{\text{proc}} u_l \lceil t_j^i(r_l)/u_l \rceil$.

For Cloud resources, two scenarios are considered: one with an unlimited number of resources per type and one with a maximum of 20 resources per type, the latter keeping the scheduling plan search space tractable [27]. Edge resources are always limited to 20 resources per type.

Cloud and Edge resources are interconnected via a dedicated link with a bandwidth of 200 Mbps. The bandwidth available for data transfers between resources and external data sources or storage devices depends on the resource layer: Edge resources leverage a bandwidth of 400 Mbps towards external data sources, while Cloud resources leverage a bandwidth of 20 Gbps towards external storage devices. When a data transfer involves two resources with different bandwidths, the smaller of the two determines the effective transfer rate. A bandwidth cost of 0.005 USD/GByte is charged for inter-layer data transfers (i.e., between Cloud and Edge resources), while intra-layer transfers carry no additional bandwidth cost.

To quantify the performance uncertainty affecting the Cloud-Edge resources, a variability factor VF is introduced. This factor describes the relative degradation of resource performance from its nominal (i.e., maximum) value. Under the probabilistic formulation, the performance parameters V_l^{proc} , $V_{l,k}^{\text{xfer}}$, V_l^{in} , and V_l^{out} are random variables sampled uniformly in the interval $[\text{nominal} \times (1 - \text{VF}), \text{nominal}]$. A variability factor of $\text{VF} = 0$ corresponds to a deterministic scenario in which all resources operate at their nominal processing capacity, while increasing values of VF introduce progressively wider fluctuations. In the experiments, values of $\text{VF} \in \{0, 0.2, 0.4, 0.6\}$ are considered.

6.1.3 SIMULATION SETTINGS

The multi-objective optimization is solved using the NSGA-II algorithm, selected for both convergence speed and ability to maintain spread and diversity among the identified solutions. The implementation is built on the jMetal 6.0 framework [163].

Binary tournament selection is used to choose parent individuals for reproduction. The crossover and mutation probabilities are set equal to 0.9 and 0.02, respectively, as determined through a sensitivity analysis based on grid search, where the mutation probability emerged as the most impactful [164]. The initial population consists of 1,000 individuals generated according to the strategies described in Section 5.4. The GA terminates after 500,000 evaluations of the fitness function.

The Monte Carlo simulation used for fitness evaluation relies on the SSJ Java library for stochastic simulation [165]. At each evaluation, the random variables modeling resource performance are repeatedly sampled to produce empirical probability distributions of the workload execution time and monetary cost. The simulation employs an adaptive stopping criterion based on the confidence interval of the standard error: trials continue until the 99% confidence interval falls below a target accuracy threshold, ensuring stable estimates of the distribution statistics.

Both objectives target the 95th percentile of their respective probability distributions.

To benchmark the proposed scheduling strategy against the state of the art, the experiments include MOHEFT as a baseline.

The scheduling plans are produced offline, before any task is executed. The experiments were run on servers with 48 to 80 logical cores (e.g., Intel Xeon Gold 6146, AMD EPYC 7F72, Intel Xeon Silver 4316 processors); the runs reported below were allocated eight logical cores each, with several runs executed concurrently on the same server. The wall-clock time of a complete run of the GA is about 22 minutes on the 26-task synthetic workloads and about one hour on the 104-task ones; quadrupling the number of tasks increases the wall-clock time by a factor of 2.7.

The probabilistic formulation is the dominant cost factor: on the Cycles workload, the wall-clock time grows from 4 to 35 hours as the variability factor VF rises from 0 to 0.6, because a deterministic fitness evaluation requires a single simulation trial, whereas a probabilistic one requires several thousand trials to satisfy the stopping criterion of the Monte Carlo simulation. These times are compatible with the offline scheduling strategy, in which the scheduling plans are determined once, before the workloads are executed.

6.2 Results

The experiments span complementary scenarios that vary the workload composition, the Cloud-Edge infrastructure configuration, the level of performance variability affecting the resources, and the presence of deadlines on high-priority workflows. Together, they quantify the benefits of allocation in the Cloud-Edge Continuum, of the interleaving of multiple workflows, and of the initial population strategies introduced in Section 5.4. The results also assess the impact of performance variability on the obtained scheduling plans.

6.2.1 BENEFIT OF THE CLOUD-EDGE CONTINUUM

The first set of experiments investigates the benefit of the Cloud-Edge Continuum with respect to provisioning on Cloud-only or Edge-only resources, on a single workflow and on multiple workflows.

The single-workflow experiment uses the balanced flavor of the IoT data processing workflow ($CCR = 1$) with aggregate input data of 15 GBytes received from external sources. Figure 6.5(a) reports the Pareto fronts obtained when allocation is restricted to Cloud-only or Edge-only resources, while Figure 6.5(b) shows the front identified when scheduling plans can leverage resources from both layers, that is, the full Cloud-Edge Continuum.

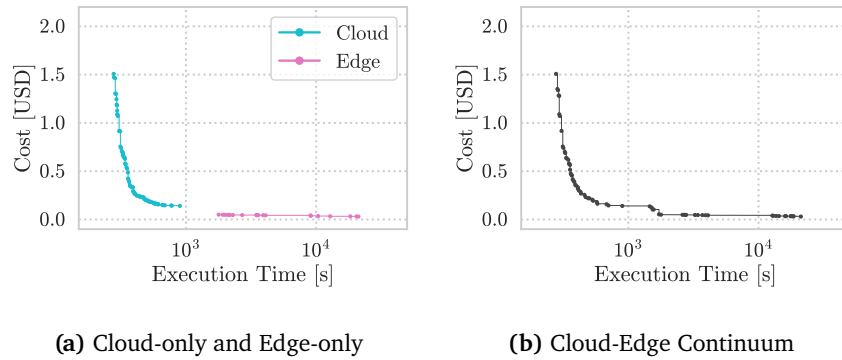


FIGURE 6.5: Pareto fronts for the balanced IoT data processing workflow obtained with: (a) Cloud-only (light blue) and Edge-only (pink) resources; and (b) the Cloud-Edge Continuum. The x -axis is in log scale.

The Cloud-only front concentrates at short execution times and high monetary cost, whereas the Edge-only front reaches much lower costs at considerably

longer execution times. A pronounced gap separates the two regions, leaving the intermediate tradeoff range uncovered by either tier in isolation. The Cloud-Edge Continuum closes this gap, producing a Pareto front that spans the entire cost and execution time range with hybrid scheduling plans combining resources from both layers.

The same conclusion extends to the scheduling of multiple workflows. Figure 6.6 shows the comparison of the Pareto fronts obtained for the four synthetic workloads $\mathcal{L}_{f_1}$, $\mathcal{L}_{2f_1}$, $\mathcal{L}_{3f_1}$, and $\mathcal{L}_{4f_1}$ when scheduling is restricted to either Cloud-only or Edge-only resources, with an unlimited number of Cloud resources per type.

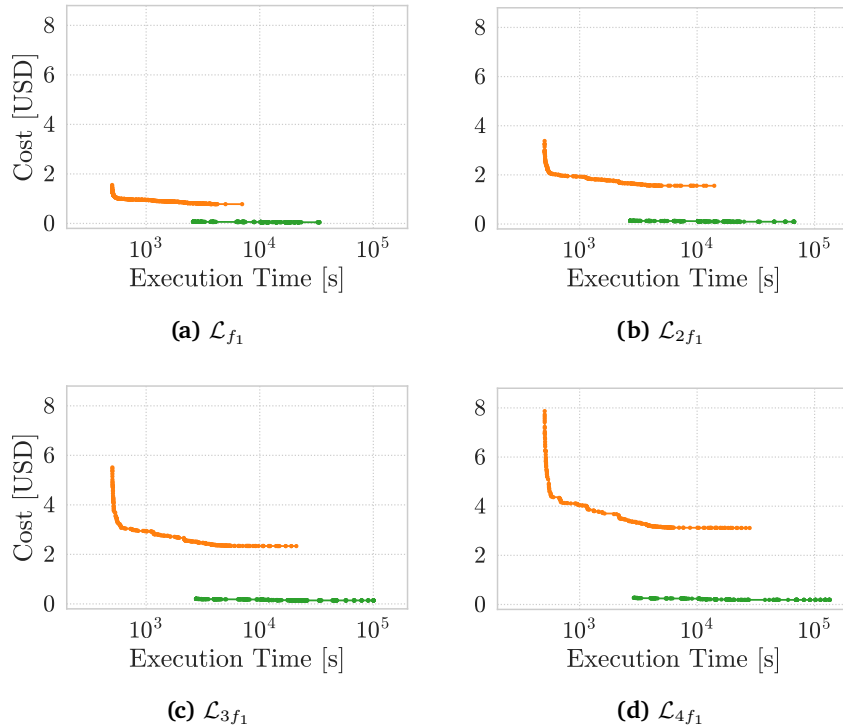


FIGURE 6.6: Pareto fronts for $\mathcal{L}_{f_1}$, $\mathcal{L}_{2f_1}$, $\mathcal{L}_{3f_1}$, and $\mathcal{L}_{4f_1}$ under Cloud-only (orange) and Edge-only (green) infrastructures. The x -axis is in log scale.

The two scenarios produce substantially different fronts. Cloud-only scheduling plans achieve much shorter execution times than their Edge-only counterparts, but at considerably higher monetary cost. Edge-only scheduling plans, conversely, attain low cost at the expense of much longer execution times. Within the execution time range covered by both fronts, the Edge-only scheduling plans are markedly cheaper than their Cloud-only counterparts at the same execution time. The gap between the two fronts widens as the workload size

grows from one to four instances of f_1 , since the limited Edge resources become fully utilized and resource contention increases.

Neither layer alone covers the full tradeoff. Scheduling plans that exploit both Cloud and Edge resources represent a good tradeoff between execution time and cost. The remaining experiments therefore consider the full Cloud-Edge Continuum.

6.2.2 BEHAVIOR ON MULTIPLE WORKFLOWS

The second set of experiments examines how the proposed framework behaves on the full Cloud-Edge Continuum as the number of workflows per workload grows and as the per-task processing demands and data volumes increase. The 12 workloads of the first synthetic set, that is, $\mathcal{L}_{nf_i}$ for $n \in \{1, \dots, 4\}$ and $i \in \{1, 2, 3\}$, are scheduled with unlimited Cloud resources and Edge resources limited to 20 per type. In addition, to compare the proposed strategy against the state of the art, MOHEFT is executed on the same workloads and infrastructure configuration.

Figure 6.7 shows that the proposed strategy generally outperforms MOHEFT: at comparable cost the scheduling plans achieve shorter execution times, and at comparable execution time they are cheaper, independently of the workload characteristics.

Within each workload, execution time decreases as monetary cost increases, with the front spanning over an order of magnitude on the cost axis. Across workloads, the minimum execution time changes only marginally as the number of workflow instances grows: for example, the minimum execution time of $\mathcal{L}_{f_1}$ is 498.5 seconds, whereas $\mathcal{L}_{4f_1}$ is less than 5.8 seconds longer despite the number of tasks being four times larger. The maximum execution times instead grow substantially with the total processing demand. Larger workloads therefore retain comparable best-case execution times while paying for the additional processing capacity through higher cost.

Each scheduling plan is analyzed in terms of the number of Cloud and Edge resources allocated. The resulting breakdown is shown in Figure 6.8 as a function of monetary cost, with the scheduling plans of each workload ordered from the cheapest to the most expensive.

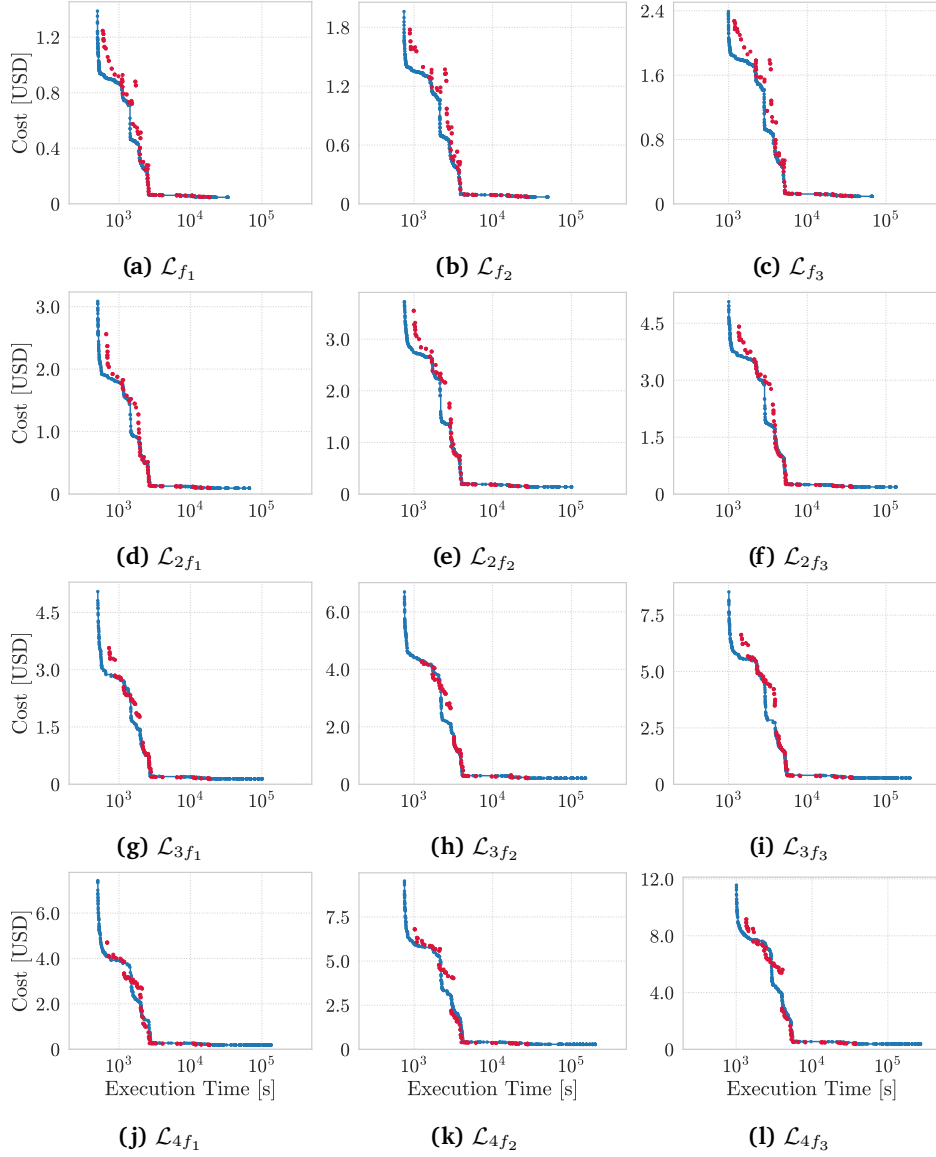


FIGURE 6.7: Pareto fronts for the 12 synthetic workloads. Blue dots refer to the scheduling plans identified by the proposed strategy, red dots to those produced by MOHEFT. The x -axis is in log scale.

It can be observed that most Pareto-optimal scheduling plans combine Cloud and Edge resources. While the cheapest scheduling plans rely exclusively on Edge resources, even the most expensive ones retain a non-negligible usage of Edge resources. Cost grows with the number of Cloud resources, and this increase is most pronounced for the more demanding workloads, where the limited number of Edge resources is no longer sufficient to absorb the (addi-

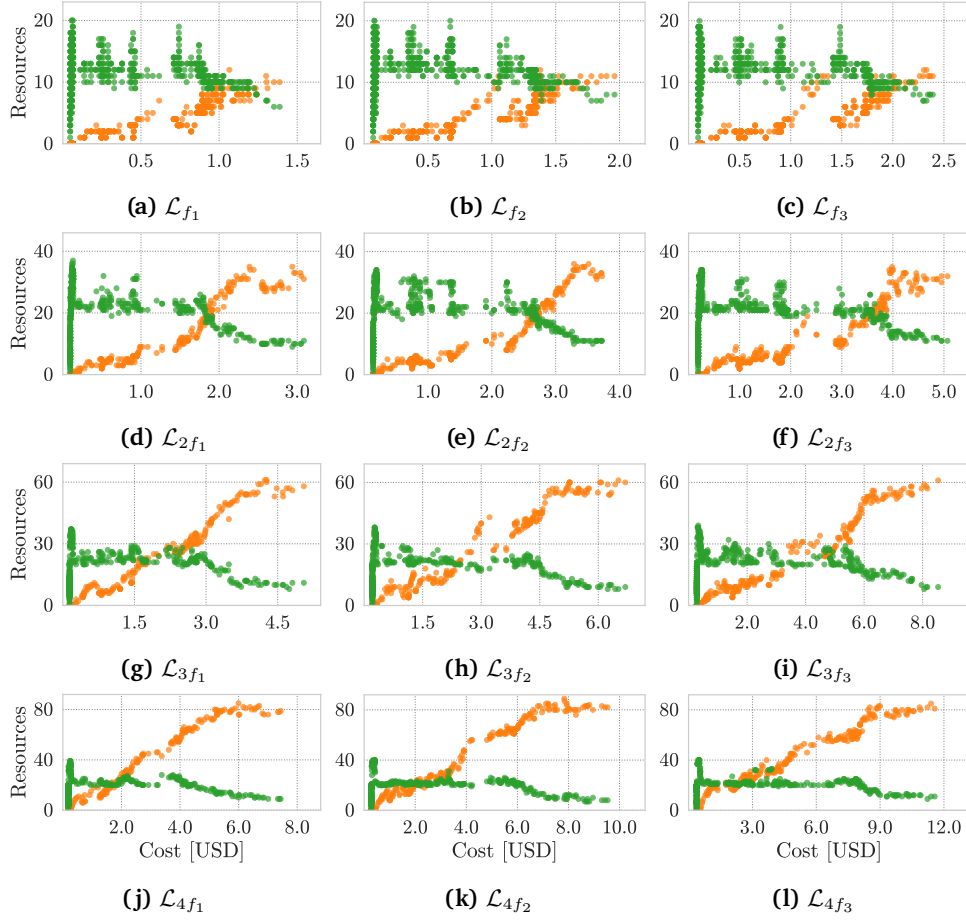


FIGURE 6.8: Number of Cloud (orange) and Edge (green) resources allocated by the optimal scheduling plans for the 12 synthetic workloads, plotted as a function of their monetary cost.

tional) processing and data transfer demands.

In the more demanding workloads, the Edge resource count decreases in steps as monetary cost grows. The two Edge types are each capped at 20 resources, so the count starts from its maximum of 40. It first drops to a plateau near 20, as the lower-capacity Edge type is fully released, and falls below 20 only for the most expensive scheduling plans, where the higher-capacity Edge type is reduced as well.

To assess the sensitivity of the results to the random components of the GA, namely the generation of the initial population and the selection, crossover and mutation operators, 30 runs of the experiment on the 12 synthetic workloads were performed, changing only the seed of the random number generator. The

minimum monetary cost is identical in all 30 runs of every workload, whereas the minimum execution time and the number of non-dominated scheduling plans vary across runs. The corresponding confidence intervals are nonetheless narrow: the 95% confidence interval of the mean minimum execution time lies within 0.6 seconds of the mean, and that of the mean number of non-dominated scheduling plans (between 415 and 698 depending on the workload) within 21. For every workload, the confidence interval of the minimum execution time falls entirely below the value achieved by MOHEFT, with a reduction between 15% and 40%. The minimum monetary cost is also lower than that of MOHEFT, by 1.1% to 1.2%. The advantage of the GA is therefore not due to a particular seed.

6.2.3 IMPACT OF SEQUENTIAL VERSUS MULTIPLE WORKFLOW EXECUTION

The benefit of scheduling multiple workflows over the sequential execution of independent workflows is quantified by comparing the scheduling of workload $\mathcal{L}_{4f_i}$ for $i \in \{1, 2, 3\}$ against four sequential runs of $\mathcal{L}_{f_i}$. Figure 6.9 shows

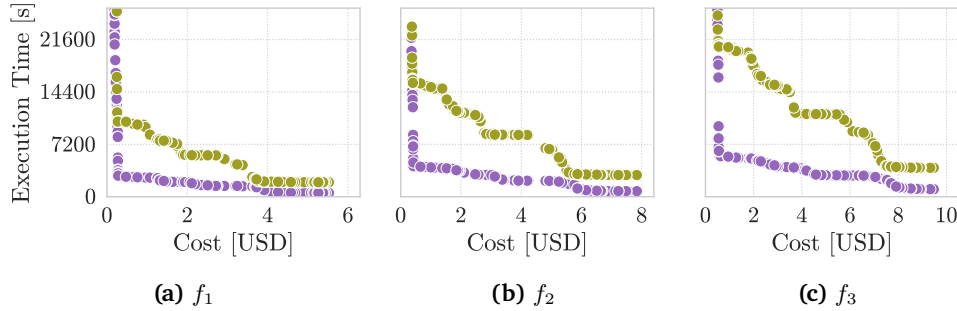


FIGURE 6.9: Execution times of four sequential standalone workflows (golden) compared to those of the same four workflows interleaved in a single workload (purple), as a function of monetary cost.

that, at any given monetary cost, the scheduling plans for executing multiple workflows achieve substantially shorter execution times than the corresponding sequential ones. For example, at a cost of 2 USD, the execution times for $\mathcal{L}_{4f_1}$, $\mathcal{L}_{4f_2}$, and $\mathcal{L}_{4f_3}$ are 34, 60, and 87 minutes respectively, whereas execution times are roughly three times longer when sequentially scheduling the four workflows. These reductions demonstrate the ability of the proposed strategy to efficiently exploit the Cloud-Edge resources by interleaving the execution of multiple workflows.

6.2.4 IMPACT OF MIXED WORKFLOWS

The second set of synthetic workloads, $\mathcal{L}_{f_1f_2}$, $\mathcal{L}_{f_1f_3}$, $\mathcal{L}_{f_2f_3}$, and $\mathcal{L}_{f_1f_2f_3}$, combines the three flavors f_1 , f_2 , f_3 into mixed workloads of two or three distinct workflows. Figure 6.10 reports the corresponding Pareto fronts.

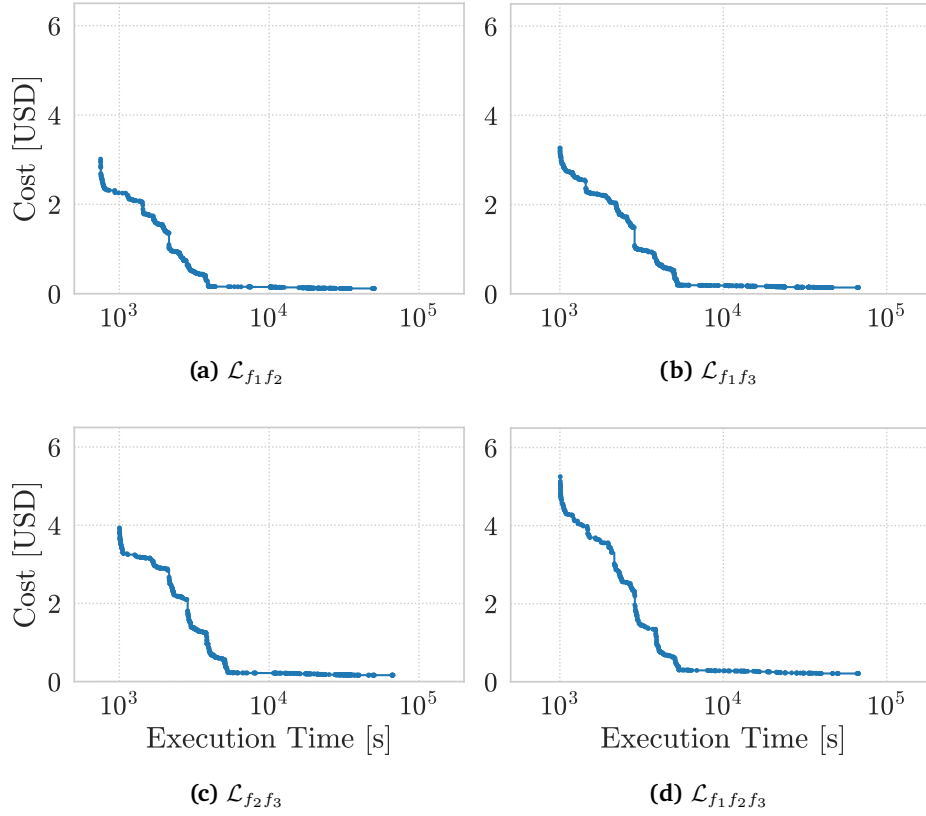


FIGURE 6.10: Pareto fronts for the four mixed-workflow synthetic workloads. The x -axis is in log scale.

The values of the two objective functions for the mixed workloads are very similar to those of their more demanding component workflow. For example, in the Pareto front of $\mathcal{L}_{f_1f_2}$, a monetary cost of 1 USD corresponds to an execution time of 36 minutes, whereas for $\mathcal{L}_{f_2}$ this cost corresponds to an execution time only one minute greater. For $\mathcal{L}_{f_1}$, the corresponding execution time is about 9 minutes.

Figure 6.11 shows that the scheduling plans for $\mathcal{L}_{f_1f_2}$, the least demanding of the mixed workloads, rely mainly on Edge resources rather than on the Cloud. Since the Edge is the lower-cost tier, serving most of the workload from it keeps

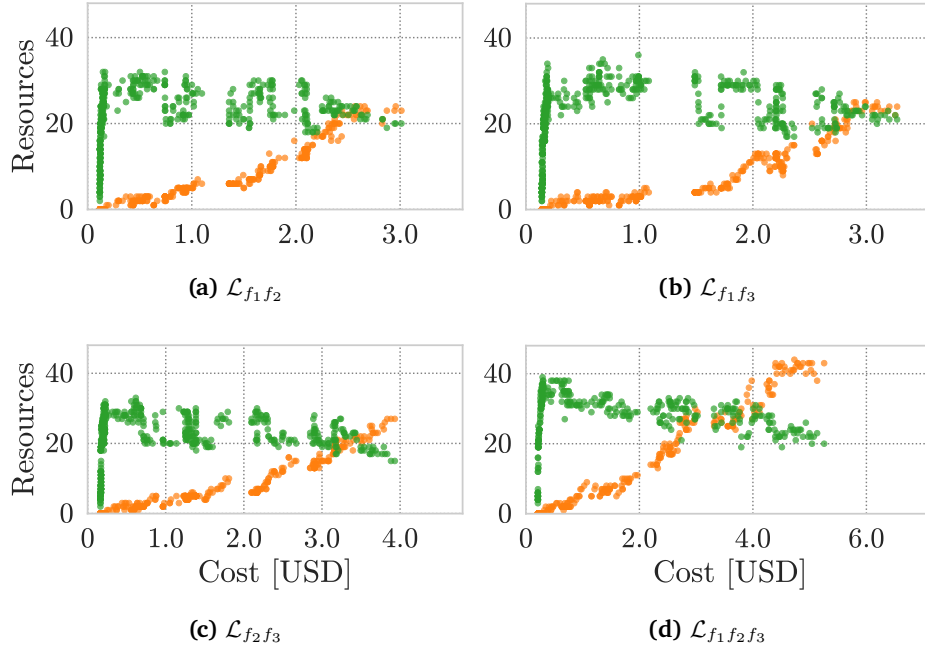


FIGURE 6.11: Number of Cloud (orange) and Edge (green) resources allocated by the optimal scheduling plans for the mixed-workflow workloads.

the monetary cost low, demonstrating the benefit of the Cloud-Edge infrastructure for scheduling multiple workflows. The proposed strategy copes well with workloads consisting of multiple workflows by adapting to the characteristics of the individual workflows.

6.2.5 EFFECT OF THE INITIAL POPULATION

The contribution of the strategies for the choice of the initial population (see Section 5.4) is assessed by solving the optimization problem with the customized initial population with respect to a randomly generated initial population. The two configurations are compared on three workloads of growing size, namely one, two, and three instances of $\mathcal{L}_{f_1 f_2 f_3}$, consisting of three, six, and nine workflows and corresponding to scheduling problems with 78, 156, and 234 tasks.

Figure 6.12 shows that with the customized version, a larger number of non-dominated scheduling plans are identified, including some with shorter execution times and comparable or substantially lower cost with respect to the baseline ones. The differences between the two fronts become more evident

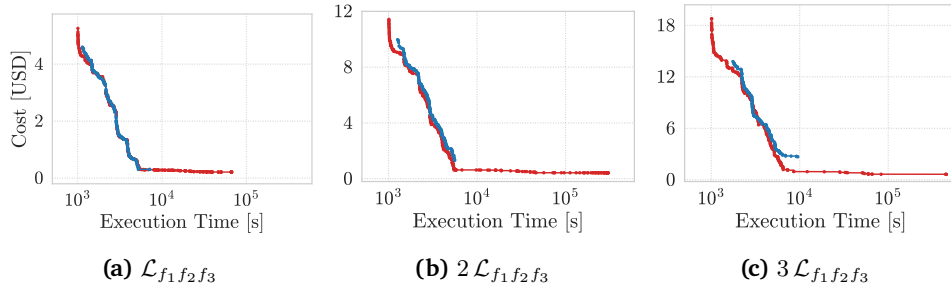


FIGURE 6.12: Pareto fronts for one, two, and three instances of $\mathcal{L}_{f_1 f_2 f_3}$ obtained with the customized initial population (red) and a randomly generated initial population (blue). The x -axis is in log scale.

as the number of tasks of the workloads grows. Overall, the customized initial population yields a better time-cost tradeoff than the random-initialization baseline across all three workloads.

For each initial population strategy, the comparison was repeated over 30 independent runs differing only in the seed of the random number generator. For all three workloads, the minimum execution time, the minimum cost and the number of non-dominated scheduling plans obtained across the 30 runs with the customized initial population improve on those obtained in the best run with the random one. The customized initial population also yields more stable results across runs: the minimum cost does not change and the minimum execution time varies by less than 1%, whereas with the random initial population both vary by more than 10%.

Table 6.5 shows the difference at the extremes of the fronts. For the largest

Workload	Customized			Random		
	Min time [s]	Min cost [USD]	Scheduling plans	Min time [s]	Min cost [USD]	Scheduling plans
$\mathcal{L}_{f_1 f_2 f_3}$	1006	0.21	594	1033	0.31	305
$2 \mathcal{L}_{f_1 f_2 f_3}$	1008	0.42	360	1493	1.30	226
$3 \mathcal{L}_{f_1 f_2 f_3}$	1008	0.63	281	2131	3.10	176

TABLE 6.5: Minimum execution time, minimum cost and number of non-dominated scheduling plans obtained with the customized and the randomly generated initial population, as medians over 30 independent runs.

workload, the customized initial population leads to a minimum execution time less than half than the one obtained with the random one, a minimum

cost about five times lower, and roughly 1.6 times as many non-dominated scheduling plans. The customization therefore allows the GA to obtain better extremes with a wider coverage of the objective space, since those fronts extend considerably further into the region of long execution times and low cost.

6.2.6 IMPACT OF PERFORMANCE VARIABILITY

The experiments under the probabilistic formulation investigate how the performance variability affecting Cloud-Edge resources influences the identified scheduling plans. The contributions of processing capacity variability and network bandwidth variability are first examined in isolation on the IoT workflow, and then jointly on the Cycles and CyberShake workloads.

The single-workflow analysis uses the computation-intensive flavor of the IoT data processing workflow ($CCR = 0.1$) scheduled on the Cloud-Edge Continuum. Variability is introduced separately in the processing capacity of the resources and in the network bandwidth. Figure 6.13 reports the corresponding Pareto fronts.

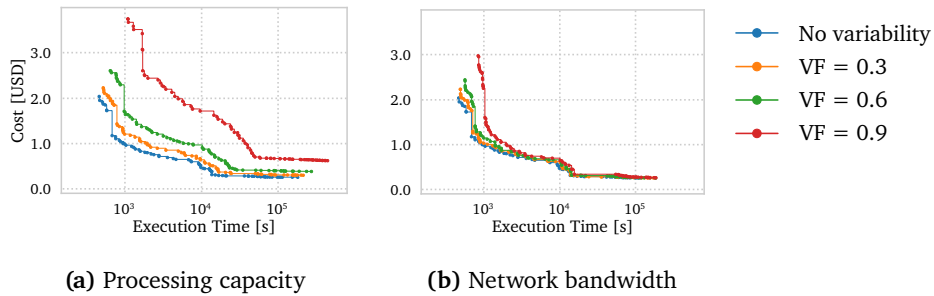


FIGURE 6.13: Pareto fronts for the computation-intensive IoT data processing workflow as a function of the variability introduced separately in (a) the processing capacity and (b) the network bandwidth of the infrastructure. The x -axis is in log scale.

Figure 6.13 shows that, as expected, the variability of the processing capacity strongly affects the Pareto fronts, whereas the variability of the network bandwidth has more limited effects.

The next experiments switch to the Cycles ($\mathcal{L}_A$) and CyberShake ($\mathcal{L}_S$) workloads, applying a single variability factor VF to both processing and network parameters. Both Cloud and Edge resources are now limited to 20 resources per type, reflecting the leasing budget constraints of production environments.

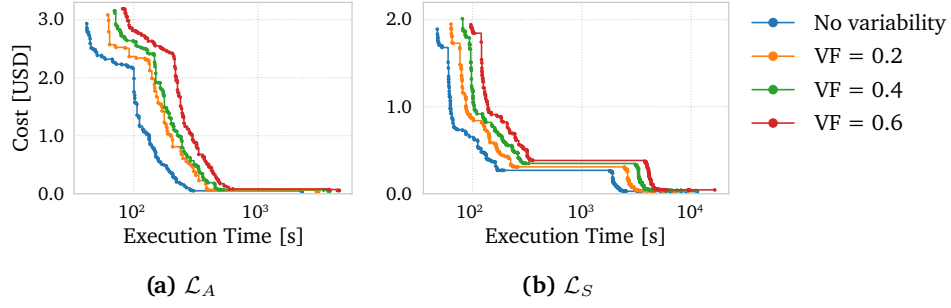


FIGURE 6.14: Pareto fronts for the Cycles ($\mathcal{L}_A$) and CyberShake ($\mathcal{L}_S$) workloads as a function of the variability factor VF. The x -axis is in log scale.

Figure 6.14 shows that both execution times and monetary costs grow with VF. At $\text{VF} = 0.6$, the shortest execution times for the Cycles and CyberShake workloads are 82 and 96 seconds respectively, roughly twice those obtained without variability, while their costs remain comparable.

For a given execution time, the cost gap widens with VF: at an execution time of approximately 210 seconds for the Cycles workload, the monetary cost grows from 0.28 USD at $\text{VF} = 0$ to 2.4 USD at $\text{VF} = 0.6$, an order of magnitude.

6.2.7 IMPACT OF DEADLINE CONSTRAINTS

To reproduce a scenario in which a high-priority workflow must complete within a pre-defined time frame, a deadline $d_{f_{A_1}} \in \{20, 40, 60\}$ seconds is imposed on f_{A_1} , the least demanding of the three Cycles flavors composing $\mathcal{L}_A$, while the workload as a whole is optimized for execution time and cost. The other two flavors remain unconstrained. The variability factor VF takes again the values 0, 0.2, 0.4, and 0.6.

Figure 6.15 shows that tighter deadlines push the front toward higher cost, since the most powerful and expensive resources must be allocated in larger numbers to satisfy the deadline. The shift becomes more severe as VF grows, because the percentile-targeted formulation must account for the worst-case execution times of the constrained workflow.

Table 6.6 reports the resource breakdown of three scheduling plans, all referring to an execution time of approximately 200 seconds at $\text{VF} = 0.2$, under the three deadlines. The 20-second deadline drives the allocation toward 18 Cloud-4XLarge resources, while the 40- and 60-second deadlines require only three and two of them, respectively. The cheaper Cloud and Edge resource

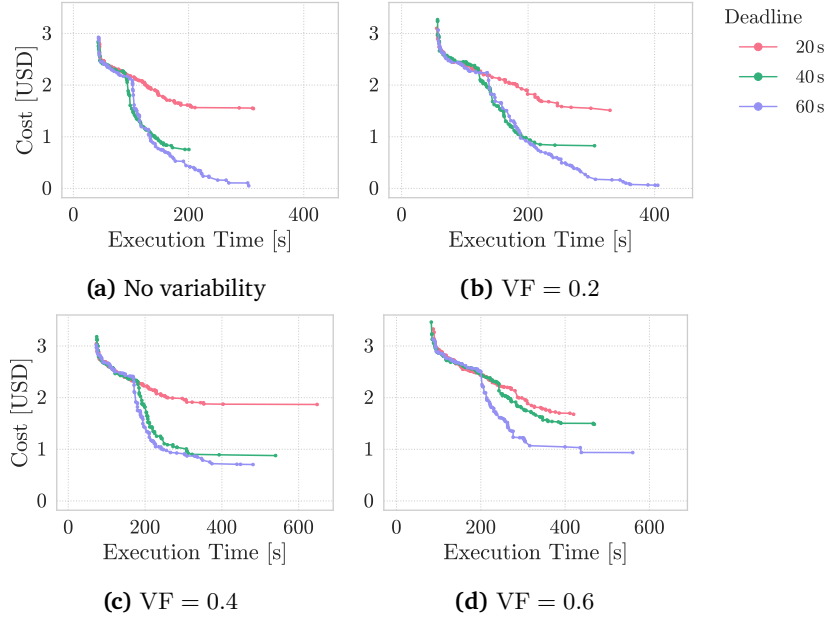


FIGURE 6.15: Pareto fronts for the Cycles workload $\mathcal{L}_A$ under three deadlines on workflow f_{A_1} and four variability factors.

counts are largely unchanged across the three scheduling plans. The corresponding monetary costs are 1.82 USD for the 20-second deadline against 0.93 and 0.88 USD for the looser ones.

The final experiment investigates how the workflow chosen as deadline-constrained affects the Pareto front of the CyberShake workload. The deadline is fixed at 70 seconds, and is applied to one of the three flavors f_{S_1} , f_{S_2} , f_{S_3} at a time, with $VF \in \{0.2, 0.4, 0.6\}$.

$d_{f_{A_1}}$ [s]	Cloud				Edge	
	4XLarge	2XLarge	Large	Medium	Large	Small
20	18	0	13	17	20	17
40	3	3	10	20	20	16
60	2	6	10	20	20	14

TABLE 6.6: Cloud and Edge resource counts for three scheduling plans of the Cycles workload that refer to an execution time of approximately 200 seconds, as a function of the deadline on its least demanding flavor, with a variability factor of 0.2.

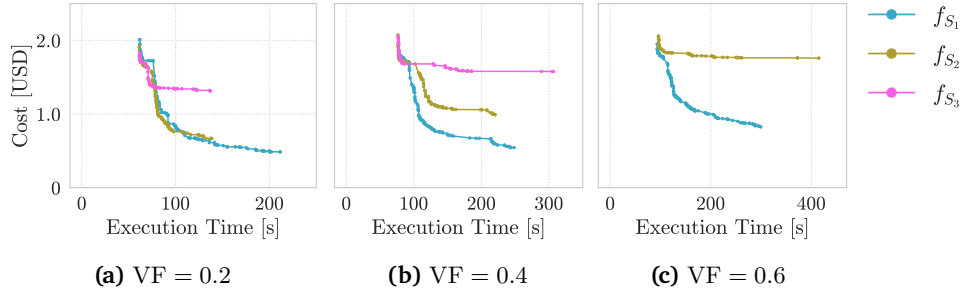


FIGURE 6.16: Pareto fronts for the CyberShake workload $\mathcal{L}_S$ obtained by constraining one workflow at a time with a deadline of 70 seconds, for three variability factors.

Figure 6.16 shows that the Pareto front depends strongly on which workflow is constrained. Constraining workflow f_{S_3} , whose processing demand is the largest among the three flavors, produces the most expensive scheduling plans, since the deadline can be satisfied only by allocating the most powerful resources. The cost penalty grows with VF, and at VF = 0.6 the GA no longer identifies any feasible scheduling plan for the constrained f_{S_3} . Constraining f_{S_1} or f_{S_2} instead always yields feasible scheduling plans, although their time-cost tradeoffs degrade with VF.

CHAPTER 7

AUTOSCALING FRAMEWORK

This chapter presents the detailed solution approach of the predictive autoscaling framework introduced in Section 4.4.

The framework combines a workload predictor for forecasting the incoming arrival rate, a capacity estimator for determining the required capacity to handle the predicted load, and the scaling policies for issuing the corresponding provisioning actions.

7.1 Workload Predictor

The workload predictor forecasts the future arrival rate, allowing scaling decisions to take effect before the predicted demand materializes.

Among the available techniques (e.g., linear regression, polynomial regression, Auto-Regressive Integrated Moving Average, recurrent neural networks), polynomial regression is adopted as its fitting reduces to a small least-squares problem with minimal overhead.

More precisely, the model is fitted on historical arrival rates λ_i observed at M equally spaced time slots t_i ($i = 1, \dots, M$) of duration δ . The arrival rate $\hat{\lambda}_j$ at the future slots t_j ($j = M + 1, \dots, M + h$) is then predicted, where h is the time horizon, i.e., the number of time slots ahead for the prediction, set equal to the VM boot time.

In particular, the arrival rate is modeled as a polynomial function of time:

$$\hat{\lambda}_i = a_0 + a_1 t_i + a_2 t_i^2 + \dots + a_d t_i^d$$

where the coefficients $a_0, a_1, \dots, a_d$ are obtained by least-squares fitting on the historical data. Figure 7.1 illustrates the procedure in an example where five

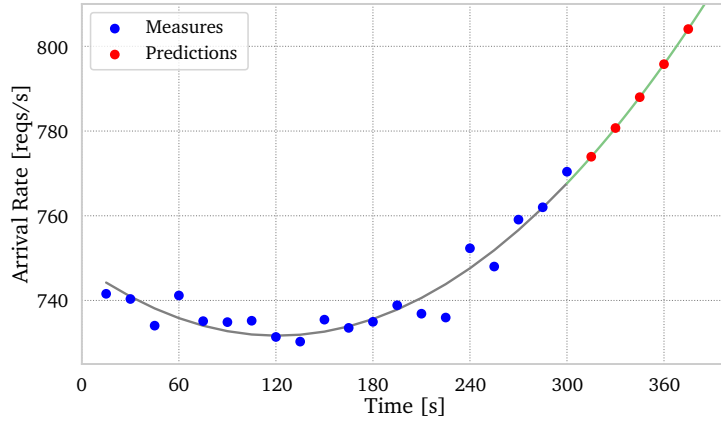


FIGURE 7.1: Example of a polynomial model fitted to a window of arrival rates, with forecasts beyond the fitting window.

minutes of a workload are sampled every $\delta = 15$ seconds, yielding $M = 20$ arrival rate measurements (blue dots), fitted with a polynomial of degree two.

The fitted model is then evaluated at later time slots to produce the forecasts $\hat{\lambda}_j$ (red dots).

Model fitting involves normalizing the time values to ensure numerical stability of the least-squares method. In particular, the time values are normalized in the interval $[0, 1]$ to ensure that all powers of t remain bounded by one.

The model is continuously updated by using a rolling window of the M most recent measurements of the arrival rate. When a new measurement is available, the oldest one is discarded and the polynomial coefficients are recomputed on the updated window. Figure 7.2 illustrates the procedure across three consecutive time slots for $M = 6$, with the latest observation entering the window at each slot highlighted in green.

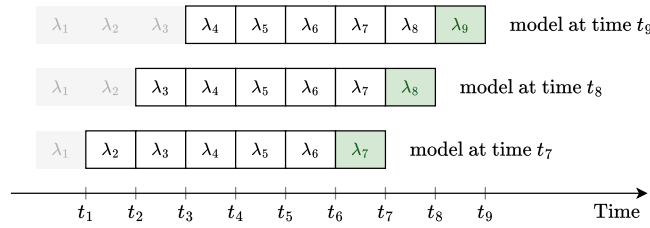


FIGURE 7.2: Rolling window update of the workload predictor across three consecutive time slots, with six arrival rate observations per window.

The size of the window M governs a trade-off between responsiveness and stability. A smaller window makes the predictor more sensitive to recent fluctuations, enabling faster adaptation to abrupt load changes but increasing the influence of noise. A larger window smooths out short-term variability and yields more stable forecasts, at the cost of slower adaptation to trend shifts. The choice of M is therefore a tuning parameter whose value is determined experimentally. Regardless of the window size, the predictor assumes that the arrival rate follows a pattern that can be approximated by a low-order polynomial. When the arrival rate is highly volatile, a low-order polynomial cannot capture its variations within the window and the forecasts can change appreciably between consecutive invocations; the policy then follows the arrival rate through more frequent scaling actions.

7.2 Resource Selector

The resource selector translates the arrival rate forecasts $\hat{\lambda}$ into provisioning actions. By applying the request backlog propagation procedure described in Section 4.4, it predicts the expected utilization of the allocated resources under the incoming workload and compares it against the desired level U^{ref} , determining whether additional capacity is required to absorb excess demand or whether the allocation can be reduced.

Two policies act on this estimate: scale-out adds resources when demand is expected to exceed the available capacity, while scale-in releases resources once they become unnecessary.

7.2.1 SCALE-OUT POLICY

Since the resources being allocated at slot i may only become available at slot $i + h$, the scale-out policy predicts the required number of resources $\hat{R}_{i+h}$ at the horizon.

The pseudo-code in Algorithm 3 formalizes the method for predicting the required number of resources used in the scale-out policy. It can be observed, at

Algorithm 3 Resource prediction for the scale-out policy

Require: i : current slot at time t_i
Require: W_i : rolling window of the last M measured arrival rates
Require: Q_i : measured backlog of requests at time t_i
Require: R : number of Cloud resources allocated at time t_j , ($j = i, \dots, i + h$)
Require: h : number of time slots ahead for the prediction
Require: S : request service demand
Require: U^{ref} : desired utilization level
Ensure: $\hat{R}_{i+h}$: The number of required Cloud resources at time t_{i+h}

- 1: $\text{arrivalModel.fit}(W_i)$ ▷ Workload prediction
- 2: $\hat{\lambda}_{i+h} \leftarrow \text{arrivalModel.predict}(t_{i+h})$
- 3: $\hat{Q}_{i+h} \leftarrow \text{BACKLOGPROPAGATION}(i, h, \text{arrivalModel}, Q_i, R, S)$
- 4: $\hat{N}_{i+h} \leftarrow \hat{Q}_{i+h} + \hat{\lambda}_{i+h} \delta$ ▷ Requests to be processed at the horizon
- 5: $\hat{X}_{i+h} \leftarrow \hat{N}_{i+h} / \delta$ ▷ Predicted throughput
- 6: $\hat{U}_{i+h} \leftarrow \hat{X}_{i+h} S$ ▷ Predicted utilization
- 7: $\hat{R}_{i+h} \leftarrow \lceil \hat{U}_{i+h} / U^{\text{ref}} \rceil$ ▷ Required number of resources

line 6, how the predicted utilization lies at the core of the policy, computed

through the utilization law (see Section 4.4) from the request service demand S and the predicted throughput $\hat{X}_{i+h}$.

However, while the service demand is directly derived from the processing capacity of the resources, the throughput is not: steady-state conditions cannot be assumed at the horizon since the allocated resources may be insufficient to process all incoming requests within a time slot.

Hence, to estimate the throughput, the predicted arrival rate as well as the backlog of unprocessed requests are first computed at the time horizon at line 2 and line 3, respectively.

The number of resources $\hat{R}_{i+h}$ required to keep the resource utilization at the desired level is then obtained at line 7 as the ratio between the predicted utilization $\hat{U}_{i+h}$ and the desired utilization level U^{ref} rounded up to the nearest integer.

The pseudo-code of the procedure used to estimate the backlog of unprocessed requests $\hat{Q}_{i+h}$ at the horizon is given in Algorithm 4.

Algorithm 4 Backlog propagation

Require: Q_i : measured backlog at time t_i
Require: $\hat{\lambda}$: predicted arrival rate
Require: R : number of Cloud resources allocated at time t_j , ($j = i, \dots, i + h$)
Require: δ : time slot duration
Require: S : request service demand
Require: h : number of time slots ahead for the prediction
Ensure: $\hat{Q}_{i+h}$: predicted backlog at time t_{i+h}

```

1: function BACKLOGPROPAGATION( $Q_i, \hat{\lambda}, R, S, \delta, h$ )
2:    $\hat{Q}_j \leftarrow Q_i$ 
3:   for  $j = i + 1$  to  $i + h$  do
4:      $\hat{N}_j \leftarrow \hat{Q}_{j-1} + \hat{\lambda}_j \delta$                                 ▷ Requests to be processed
5:      $\hat{C}_j \leftarrow (R_j / S) \delta$                                     ▷ Maximum processable requests
6:     if  $\hat{N}_j > \hat{C}_j$  then
7:        $Q_j \leftarrow \hat{N}_j - \hat{C}_j$                                 ▷ Backlog propagation
8:     else
9:        $Q_j \leftarrow 0$ 
10:    end if
11:  end for
12:  return  $\hat{Q}_{i+h}$ 
13: end function

```

The procedure operates in discrete steps, starting from the measured backlog Q_i at the time slot i the autoscaler is invoked (line 2). It iterates over the future time slots $j = i+1, \dots, i+h$, by computing the number of unprocessed requests $\hat{N}_j$ (line 4) at each slot as the sum of the backlog $\hat{Q}_{j-1}$ propagated from the previous slot and the predicted arrivals $\hat{\lambda}_j \delta$ in the current slot.

$\hat{N}_j$ is then compared against the maximum number of requests $\hat{C}_j$ that can be processed by the allocated resources during the slot, thereby updating the backlog $\hat{Q}_j$ (line 7): if the allocated resources can absorb all pending requests, the backlog vanishes; otherwise, the surplus propagates to the next slot. The final value $\hat{Q}_{i+h}$ is the predicted backlog at the horizon.

A critical aspect concerns the accounting of in-progress provisioning actions in R_j . The quantity reflects not only the currently allocated resources but also those whose provisioning was requested at earlier slots and whose boot is expected to complete by time slot j . A scale-out decision issued at slot $i-2$, for instance, provisions resources that become available at slot $i+h-2$, and these count toward R_{i+h} . In practice, the autoscaler keeps track of the expected allocation timeline by recording for each provisioning request the time the VM is expected to become available, which is equal to the request slot plus h ; newly requested resources are counted as available only once h slots have elapsed.

7.2.2 SCALE-IN POLICY

In the context of scale-in actions, the resource selector adopts a more conservative approach that avoids "early" resource deallocation. Service providers often tolerate some over-provisioning to maintain QoS. Consequently, scale-in requires the following conditions to hold across every time slot within the considered time horizon:

1. no scale-out action is in progress;
2. the predicted utilization remains below the desired level U^{ref} ;
3. the system exhibits persistent over-provisioning.

The third condition is verified by computing the largest number of resources that can be released without pushing the predicted utilization above U^{ref} at any slot within the horizon. This quantity is denoted ΔR^{over} and equals the smallest excess between the resources R_j allocated at slot j and the predicted $\hat{R}_j$ across all future slots:

$$\Delta R^{\text{over}} = \min_{j=i+1, \dots, i+h} (R_j - \hat{R}_j)$$

Taking the minimum binds the deallocation to the most demanding slot in the horizon: releasing any larger amount would leave that slot under-provisioned.

Algorithm 5 reports the procedure for computing ΔR^{over} . At each future slot,

Algorithm 5 Number of overprovisioned resources

Require: i : current slot at time t_i
Require: W_i : rolling window of the last M measured arrival rates
Require: Q_i : measured backlog of requests at time t_i
Require: R : number of Cloud resources allocated at time t_j , ($j = i, \dots, i + h$)
Require: h : number of time slots to be checked for over-provisioning
Require: δ : time slot duration
Require: S : request service demand
Ensure: ΔR^{over} : number of resources safe to deallocate

- 1: $\text{arrivalModel.fit}(W_i)$
- 2: $\Delta R^{\text{over}} \leftarrow +\infty$
- 3: **for** $j = i + 1$ **to** $i + h$ **do**
- 4: $\hat{\lambda}_j \leftarrow \text{arrivalModel.predict}(t_j)$
- 5: $\hat{N}_j \leftarrow \hat{Q}_{j-1} + \hat{\lambda}_j \delta$
- 6: $\hat{X}_j \leftarrow \hat{N}_j / \delta$
- 7: $\hat{U}_j \leftarrow \hat{X}_j S$
- 8: $\hat{R}_j \leftarrow \lceil \hat{U}_j / U^{\text{ref}} \rceil$
- 9: $\Delta R^{\text{over}} \leftarrow \min(\Delta R^{\text{over}}, R_j - \hat{R}_j)$
- 10: $Q_j \leftarrow 0$ ▷ Under the assumption that no resources are saturated
- 11: **end for**

the predicted $\hat{R}_j$ is derived through an approach similar to the one used in the scale-out policy: from the predicted arrival rate $\hat{\lambda}_j$, the number of requests $\hat{N}_j$ to be processed in the slot, the predicted throughput $\hat{X}_j$, and the predicted utilization $\hat{U}_j$, the required resource count is obtained as $\hat{R}_j = \lceil \hat{U}_j / U^{\text{ref}} \rceil$. The excess $R_j - \hat{R}_j$ is then evaluated and the minimum updated.

In contrast to the scale-out policy, the requests left unprocessed at the end of each slot are not carried over to the next one, and the backlog is assumed to be zero: since deallocation is only considered when persistent over-provisioning is expected, no resource is saturated and no backlog accumulates from one slot to the next.

The zero-backlog assumption is however limited to a single invocation: at the next time slot the scale-out policy starts again from the backlog measured

on the system, so a burst arriving after a scale-in has been decided is detected within one slot and additional resources are requested regardless of any scale-in or cool-down in progress.

When ΔR^{over} is positive, the resource selector triggers a scale-in by deallocating ΔR^{over} resources; otherwise, the allocation is left unchanged.

7.3 Heterogeneous Resource Selection

When VMs span multiple types, the resource selector no longer issues a single desired resource count but identifies a mix of VM types that meets the required capacity. However, the identification of the correct mix introduces a combinatorial problem, since the same required capacity can be obtained from many combinations of resource types with different processing capacities and leasing costs.

The resource selector addresses this through an offline enumeration of candidate mixes of heterogeneous VMs, called buckets. A bucket is a combination with repetition of up to B VMs drawn from the K available types, described by the per-type counts $R_1, \dots, R_k$ subject to $\sum_{k=1}^K R_k \leq B$. Each bucket is characterized by two aggregate values for processing capacity and leasing cost, computed as the sum of the contributions of the included VMs.

The total number of buckets is $\sum_{i=1}^B \binom{K+i-1}{i}$, obtained by summing, for each bucket size i from 1 to B , the number of size- i combinations with repetition from the K types. Because this set depends only on the resource catalog and on B , it is built before the autoscaler begins operating and reused at every invocation.

At runtime, buckets are selected by taking into account the current resource allocation. By comparing the processing capacity of the VMs allocated at time slot $i + h$ with the predicted processing capacity, the autoscaler determines whether and to what extent scaling should be performed.

For scale-out, resources are selected from the bucket with the smallest capacity and minimum cost that ensures the workload requirements. In the case of a scale-in action, the mix of VMs to be deallocated is selected according to their capacity and cost starting from the most powerful and expensive until the desired capacity is reached.

It is remarked that when heterogeneous resources are allocated, the incoming requests are distributed across the allocated resources through a weighted round-robin policy.

CHAPTER 8

EXPERIMENTAL EVALUATION OF AUTOSCALING

The proposed Load-Aware Predictive autoscaling framework described in Chapter 7 is evaluated in simulations leveraging synthetic and real workloads. Three baseline policies provide the points of comparison; namely one reactive and two static. For each experiment, both the service quality delivered to end users through the response time of the requests and the cost incurred by the provider through the total VM leasing time are quantified.

8.1 Experimental Setup

The experimental setup considers different scenarios under varying workload, Cloud infrastructure, and framework configuration.

8.1.1 WORKLOADS

Two types of workloads are considered. Synthetic workloads are used to stress the proposed autoscaling framework under distinct, time-varying arrival rates, while real workloads are used to test the framework under the fluctuating demand observed in deployed services.

SYNTHETIC WORKLOADS

Three synthetic workloads are considered, each characterized by a different time-varying arrival pattern, shown in Figure 8.1. The *increasing* workload (Figure 8.1a) is characterized by an arrival rate that grows linearly from 100 to 4,500 requests per second over a one-hour interval. The *periodic* workload (Figure 8.1b) consists of two periods of a sinusoidal pattern, each ranging from 100 to 3,600 requests per second, over a four-hour interval. The *unpredictable* workload (Figure 8.1c) does not follow any predefined pattern and is characterized by an arrival rate varying between 1,600 and 3,850 requests per second over a two-hour interval. Their main characteristics are summarized in Table 8.1.

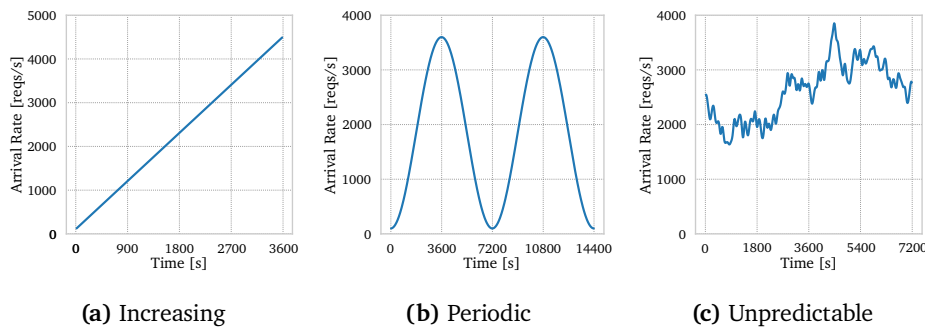


FIGURE 8.1: Arrival patterns of the synthetic workloads.

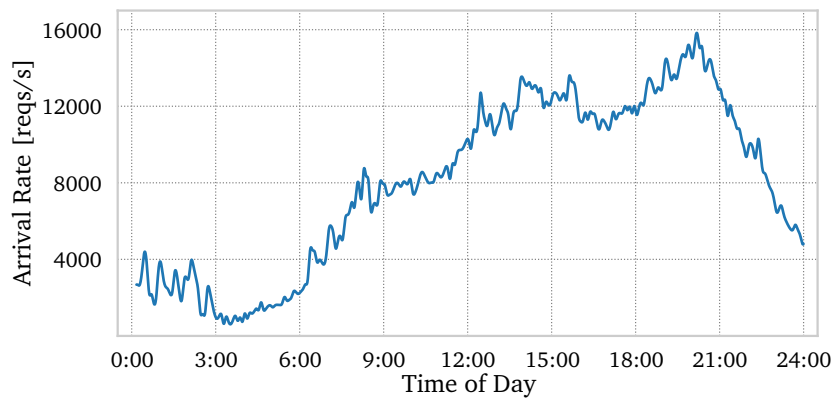
For all three synthetic workloads, requests are homogeneous, with a processing demand of 10 MI for each request.

Workload	Arrival rate		Number of requests	Duration [hours]
	Min [reqs/s]	Max [reqs/s]		
Increasing	100	4,500	8×10^6	1
Periodic	100	3,600	26×10^6	4
Unpredictable	1,600	3,850	19×10^6	2

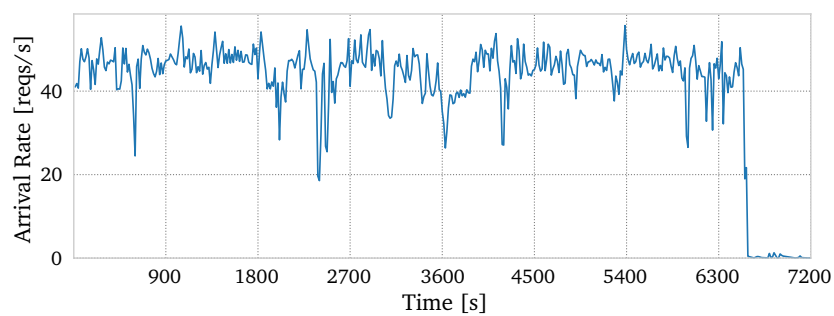
TABLE 8.1: Characteristics of the synthetic workloads considered in the experiments.

REAL WORKLOADS

The two real workloads provide realistic arrival dynamics drawn from operational systems.



(a) Flight seats availability service



(b) Amazon Redset

FIGURE 8.2: Arrival patterns of the real workloads.

The first (Figure 8.2a) is a 24-hour trace collected from a flight seats availability service [166], obtained by monitoring a set of consolidated VMs running on a single physical machine and sharing all its hardware and software resources. The arrival rate follows a diurnal pattern ranging from 600 to 15,800 requests per second, for a total of about 693 million requests. Request processing demands span 6.3 to 15.7 MI; with about 99.5% of the requests in the 8 to 14 MI range.

The second real workload (Figure 8.2b) is drawn from the Amazon Redshift dataset [167], which collects three months of user-query metadata from a sample of clusters in the Amazon Redshift fleet. The chosen pattern refers to the requests received by cluster 96 during two hours from 17:30 on March 5, 2024, for a total of 296,015 requests at an average rate of 41.11 requests per second. As can be seen from the figure, the pattern is characterized by an initial high-load phase with several fluctuations and a sudden sharp drop, and is representative of the daily behavior of the cluster over the three-month observation period. Each request is assumed to process 1,000 MI on the simulated infrastructure.

8.1.2 CLOUD INFRASTRUCTURE

Two infrastructure configurations are considered. The first one consists of homogeneous resources and serves as the baseline for the evaluation of the predictive behavior of the framework. It provides an unlimited number of VMs with processing capacity $\mu = 1,000$ MIPS. The total VM leasing time is proportional to the monetary cost and is therefore adopted as the cost metric.

The second infrastructure consists of heterogeneous resources and exercises the bucket-based selection mechanism described in Section 7.3. It provides four VM types with distinct processing capacity and leasing cost with up to twenty resources allocatable per type. Its characteristics are summarized in Table 8.2. Since leasing costs differ across VM types, leasing time is no longer proportional to monetary cost; the cost metric is therefore obtained by weighting the leasing time of each VM by the per-hour rate p_k of its type. Candidate buckets are annotated with their cost under a minimum leasing duration of two minutes, equal to the boot time plus a minimum allocation time of 60 seconds enforced after provisioning.

Type	Cores V_k	Processing capacity μ_k [MIPS]	Leasing cost p_k [USD/hr]	Max $R_k^{\max}$
Small	1	500	0.06	20
Medium	2	1,000	0.11	20
Large	4	1,600	0.14	20
XLarge	4	2,000	0.20	20

TABLE 8.2: Characteristics of the heterogeneous VM catalog used in the experiments.

The boot time is equal to 60 seconds, corresponding to $h = 4$ time slots given the invocation interval $\delta = 15$ seconds.

It is remarked that, in both configurations, the deallocation of a VM is delayed by the time required to complete the requests being processed on the VM being removed.

8.1.3 FRAMEWORK AND SIMULATION SETTINGS

The evaluation relies on a simulator built on top of the CloudSim toolkit [168], a standard tool for modeling and simulating Cloud computing environments. In particular, CloudSim is extended with the Broker, Scheduler, Autoscaler, and Monitor components required to implement the proposed framework and the baselines. The architecture is illustrated in Figure 8.3.

The Broker is the central component of the simulator and orchestrates the other elements: it receives requests from the incoming workload and forwards them to the VMs in the datacenter selected by the Scheduler according to a weighted round-robin policy.

The Autoscaling component executes, at each control cycle, the scaling algorithms of Section 7.2 or a baseline policy, and notifies scaling decisions to the Broker, which in turn operates on the datacenter.

The Monitor component records request arrivals and completions, resource allocations and deallocations, and the times at which they occur. This information is both used by the Broker to make informed decisions and written into a log file, used for offline analysis of the behavior of the framework and computation of the performance metrics.

In the experiments, the autoscaler is executed every $\delta = 15$ seconds. At each execution, the predictor is fitted on a rolling window of 20 arrival rate observa-

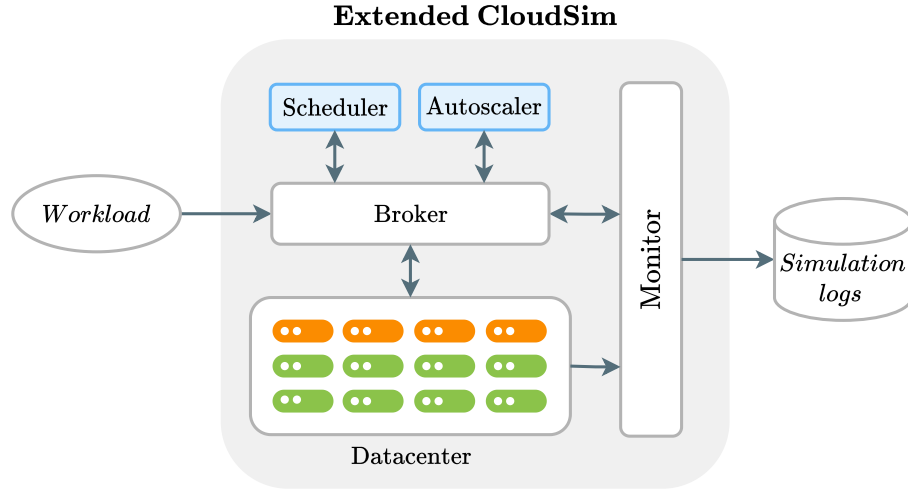


FIGURE 8.3: Architecture of the autoscaling simulation environment.

tions covering the most recent 300 seconds, and produces a workload forecast at a horizon of 4 time slots matching the 60 seconds boot time of the VMs.

The desired utilization level is set to $U^{\text{ref}} = 0.8$, and a minimum of two VMs is maintained active regardless of the predicted demand; in the heterogeneous configuration, the smallest type VMs are selected. The first 600 seconds are discarded from each simulation as warm-up to exclude transient effects in the collected measurements.

The polynomial degree of the predictor (see Section 7.1) is chosen separately for each workload by applying forward nested model selection with a maximum degree of five to prevent overfitting, which would produce unstable forecasts and oscillations in the allocated resources under highly fluctuating workloads. A polynomial of degree two, that is $\hat{\lambda}(t) = a_0 + a_1t + a_2t^2$, is selected for the synthetic and flight seats availability service workloads. Instead, for the Redset workload, a polynomial of degree one (i.e., linear regression) is selected.

Table 8.3 summarizes the configuration parameters common to all experiments together with those specific to the heterogeneous setting.

Three settings are specific to the heterogeneous infrastructure. First, each bucket consists of a mix of up to $B = 10$ VMs, yielding $\sum_{k=1}^{10} \binom{4+k-1}{k} = 1,000$ candidate buckets enumerated offline before the autoscaler begins operating. Second, a cool-down time of one minute, corresponding to the minimum time between a scale-out and a subsequent scale-in, prevents oscillations in the allocation. Third, incoming requests are dispatched to allocated resources via

Scenario	Parameter	Value
<i>Common</i>	Time slot duration	15 seconds
	Prediction horizon	4 slots
	History window	20 slots
	Desired utilization level	0.8
	Minimum allocated resources	2
	Simulation warm-up time	600 s
<i>Heterogeneous</i>	Maximum VMs per bucket	10
	Cool-down / minimum allocation time	60 s

TABLE 8.3: Autoscaler and simulation parameters. The *Common* group applies to all experiments; the *Heterogeneous* group lists parameters specific to the heterogeneous configuration.

weighted round-robin with weights proportional to per-type capacity μ_k , equalizing the per-resource utilization across types as required by the capacity estimator of Section 7.3.

The effectiveness of the framework is assessed through metrics related to the cost of the provisioned VMs and to the QoS delivered to end users. The cost is measured as the total VM leasing time under the homogeneous configuration, and as the sum of the per-type leasing times weighted by the per-type rates cost p_k under the heterogeneous configuration. The QoS is expressed by the response time of each request, i.e., the time elapsed between the arrival of a request at the system and the completion of its processing; providers must keep this quantity low to satisfy the levels agreed with end users in their SLAs.

To estimate the overhead of the policy, the time spent by all its invocations and the fraction of invocations leading to a scaling action are also recorded.

8.2 Results

The evaluation proceeds in five parts. The homogeneous framework is first compared against the reactive and static baselines on the synthetic workloads, then on the real flight seats workload. A sensitivity analysis follows, varying the desired utilization level to characterize its effect on cost and response time.

The heterogeneous configuration is examined last, both in isolation and against its homogeneous counterpart.

8.2.1 COMPARISON WITH BASELINES ON SYNTHETIC WORKLOADS

The first experiment consists in evaluating the framework in the homogeneous configuration against three baseline policies, namely the reactive *Average Load* policy and the *MID* and *MAX* static provisioning strategies [100]. Average Load scales in or out whenever the average utilization of any VM over a 75 seconds window exceeds 0.8 or falls below 0.4, respectively. In turn, *MID* and *MAX* allocate a fixed number of VMs required to process the average and the maximum load of each workload, respectively.

Experiments are performed on the increasing, periodic and unpredictable synthetic workloads previously discussed. Table 8.4 reports, for each policy, the minimum and maximum number of allocated VMs observed during the simulation, the number of scaling actions, the total VM leasing time, and the mean, standard deviation, and maximum of the response time distribution. It can

Workload	Policy	VMs	Scaling actions	Leasing [min]	Response time [ms]		
		Min–Max			Mean	Std. Dev.	Max
Increasing	Load-Aware	11–57	45	1,743	10.2	0.5	26.5
	Average Load	11–50	39	1,545	10.7	1.2	36.3
	MID	29–29	–	1,450	85,162.2	111,814.7	372,721.3
	MAX	57–57	–	2,850	10.1	0.2	17.3
Periodic	Load-Aware	2–45	169	5,751	10.3	1.0	97.3
	Average Load	2–46	173	6,302	135.4	415.9	2,554.5
	MID	24–24	–	5,520	459,297.3	363,901.2	974,606.3
	MAX	45–45	–	10,350	10.1	0.3	61.6
Unpredictable	Load-Aware	21–50	213	3,825	10.3	0.7	33.7
	Average Load	31–44	13	4,179	10.1	0.4	22.7
	MID	33–33	–	3,960	2,681.5	7,139.9	32,020.4
	MAX	49–49	–	5,880	10.1	0.2	14.3

TABLE 8.4: Comparison of the performance of the Load-Aware Predictive policy and of the three baselines on the synthetic workloads.

be observed that, across the three workloads, the Load-Aware Predictive policy attains response times comparable to MAX provisioning at a substantially lower leasing time, and outperforms MID by several orders of magnitude on response time even though MID yields a slightly lower leasing time on the in-

creasing (16.8% lower) and periodic (4% lower) patterns. The response time value reflects the inability of MID to absorb the workload whenever the arrival rate exceeds its static sizing, while MAX incurs a leasing time between 1.5 and 1.8 times that of the policy with comparable response times.

For the unpredictable workload (which offers no regular pattern for the predictor to exploit), the policy remains competitive with respect to Average Load in both leasing time and response time, at the expense of 213 scaling actions against 13 to keep up with the fast-changing arrival rate.

Figure 8.4 shows the evolution of the number of allocated resources under the proposed policy and the Average Load baseline, together with the arrival rate of each workload. The number of VMs allocated by the Load-Aware Predictive pol-

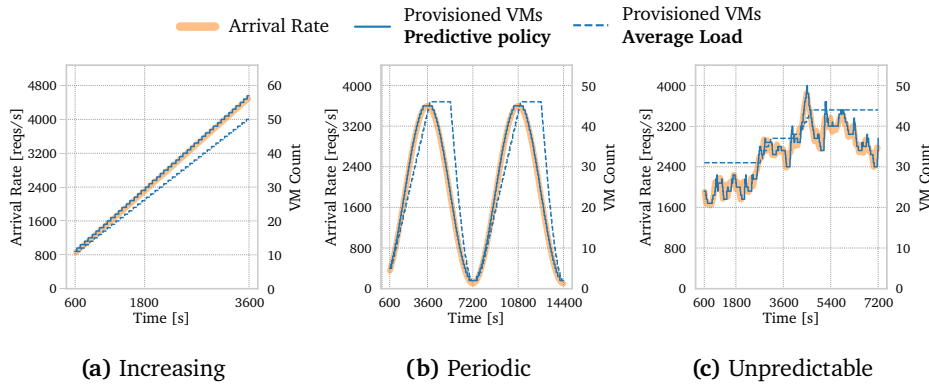


FIGURE 8.4: Workload arrival rate (yellow line) and number of VMs allocated by the Load-Aware Predictive (blue solid line) and Average Load (blue dashed line) policies on the synthetic workloads.

icy closely follows the arrival pattern of each workload, whereas Average Load lags behind: the predictor anticipates upcoming changes in the arrival rate and adjusts the allocated resources to maintain the desired utilization level, while the reactive baseline responds to past changes, resulting in delayed scaling actions, longer response times and higher VM leasing time. The effect is most pronounced on the periodic workload, where the timely scaling actions of the policy yield a leasing time about 10% lower than that of the Average Load baseline, while the delayed scale-out actions of the latter cause resource saturation that raises the mean response time by more than one order of magnitude.

It has to be remarked that the computational overhead of the policy, measured on an Intel Core i7-9700K workstation, averages about 3 ms per invocation. This is a negligible fraction of the 15 s control cycle: the entire periodic experiment completes in roughly six minutes of wall-clock time.

8.2.2 COMPARISON WITH BASELINES ON THE FLIGHT SEATS WORKLOAD

To further investigate the effectiveness of the Load-Aware Predictive policy in a realistic scenario, a second experiment evaluates it on the flight seats availability workload against the reactive Average Load baseline, using the same infrastructure. The results are summarized in Table 8.5: as for the synthetic

	Load-Aware Predictive	Average Load
Leasing time [h]	2,627	2,994
Scaling actions	1,567	362
Effective invocations [%]	27.4	6.3
Response time [ms]		
Mean	10.8	210.3
Std dev	1.4	2,813.4
Min	6.3	6.3
Max	52.3	51,657.0

TABLE 8.5: Comparison of the performance of the Load-Aware Predictive policy against the Average Load baseline on the flight seats availability workload.

workloads, the Load-Aware Predictive policy substantially outperforms the Average Load baseline. Notably, the total VM leasing time of the policy, equal to 2,627 hours, is 12% shorter than that of the baseline, thanks to the 1,567 scaling actions that timely adapt the number of allocated VMs to the diurnal pattern. The benefit is also reflected in the mean response time, one order of magnitude shorter than that of the baseline, whose slow reaction to changes in the arrival rate causes resource saturation and large response times.

8.2.3 IMPACT OF THE DESIRED UTILIZATION LEVEL

A third experiment is aimed at evaluating the impact of the desired utilization level U^{ref} on the scaling decisions. To this end, the flight seats availability workload is simulated with U^{ref} varied between 0.5 and 0.95. Low values prevent resource saturation caused by workload prediction errors or sudden arrival rate spikes, although these advantages come at the expense of a larger number of allocated VMs and longer leasing times.

Figure 8.5 reports the number of allocated VMs over the 24-hour trace for each value of U^{ref} . It can be observed that all the curves follow the diurnal pattern of

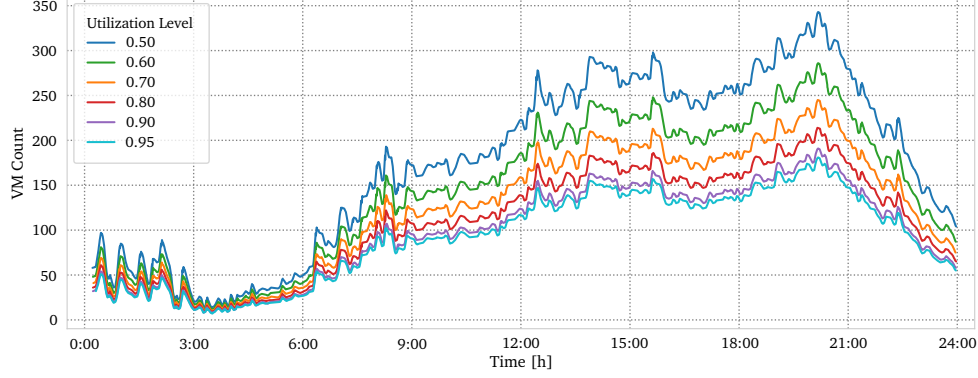


FIGURE 8.5: Number of allocated VMs over 24 hours by varying the desired utilization level on the flight seats availability workload.

the arrival rate, with their amplitude differing substantially across utilization levels. Low values of the desired utilization level lead to over-provisioning: the framework allocates up to 343 VMs at peak and 175 on average at $U^{\text{ref}} = 0.5$. On the other end, the average allocation drops to 97 and 92 VMs at $U^{\text{ref}} = 0.9$ and 0.95 , with a peak as low as 181 in the latter case, leading to under-provisioning.

The choice of U^{ref} also impacts the response time distribution, as shown in Table 8.6. To further observe the effects of the different utilization levels, Figure 8.6 plots the 99th percentile of the response time distribution as a function of the total VM leasing time, and of U^{ref} . The curve exhibits a knee at $U^{\text{ref}} = 0.8$: below this value, over-provisioning keeps response times low at the expense of a higher total leasing time, while above it the allocated resources approach saturation and the response time deteriorates rapidly. The knee value represents a good tradeoff, and is therefore adopted as the default U^{ref} in the other experiments.

8.2.4 IMPACT OF HETEROGENEOUS RESOURCES

Additional experiments evaluate the framework in the heterogeneous configuration on the Amazon Redset workload. The characteristics of the VMs are summarized in Table 8.2. The desired utilization level U^{ref} is 0.8.

Response time [ms]	Utilization Level U^{ref}					
	0.50	0.60	0.70	0.80	0.90	0.95
Min	6.3	6.3	6.3	6.3	6.3	6.3
First quartile	9.7	9.7	9.7	9.7	9.9	10.4
Median	10.7	10.7	10.7	10.8	11.2	11.8
Third quartile	11.9	11.9	11.9	11.9	12.4	13.3
90 th percentile	12.7	12.7	12.7	12.8	13.4	15.3
95 th percentile	13.1	13.1	13.1	13.2	14.1	16.9
99 th percentile	13.7	13.7	13.7	13.9	16.3	20.8
99.9 th percentile	14.1	14.1	14.3	15.8	20.7	29.1
Max	23.2	30.3	31.8	52.3	101.5	3,127.9

TABLE 8.6: Response time distribution on the flight seats availability workload as a function of the desired utilization level.

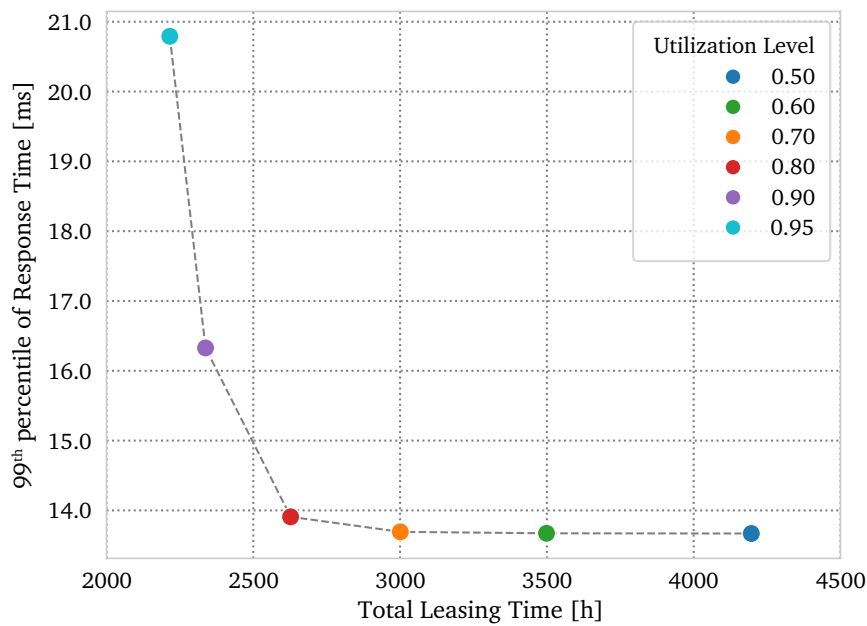


FIGURE 8.6: 99th percentile of the response time distribution as a function of the total VM leasing time for different values of the desired utilization level on the flight seats availability workload.

Table 8.7 reports, for each VM type, the average and maximum number of allocated VMs together with its contribution to the overall leasing cost on the Red-set workload. Notably, to cope with the workload fluctuations, the framework

Type	Allocated VMs		Cost [USD]
	Avg.	Max	
Small	9.33	20	1.08
Medium	17.08	20	3.30
Large	4.99	18	1.28
XLarge	13.44	20	4.74
Overall	40.86	60	10.40

TABLE 8.7: Per-type allocation statistics on the Redset workload under the heterogeneous configuration.

allocates up to 20 VMs of every type except Large, while the average number varies from 4.99 to 17.08. The per-type contribution to the total leasing cost also varies substantially across VM types.

Figure 8.7 shows the breakdown of allocated VMs by type over time as a stacked plot, where the total height corresponds to the number of allocated resources and each band to the count of one VM type. The autoscaler distributes the

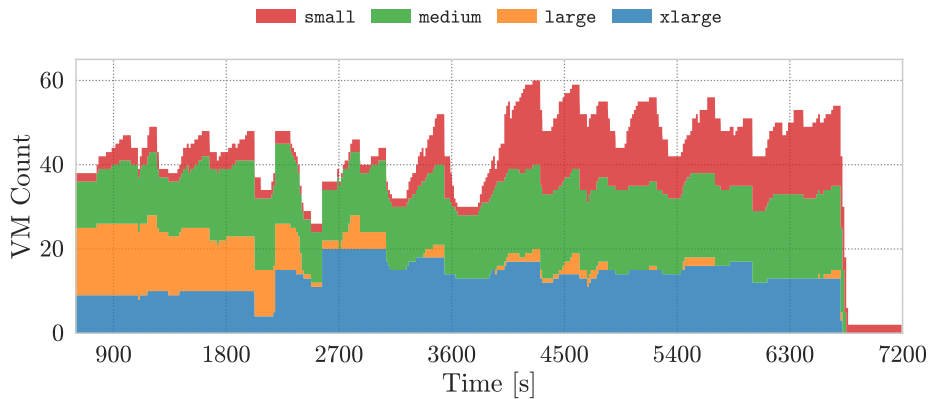


FIGURE 8.7: Breakdown of the number of VMs allocated by the autoscaler on the Redset workload.

allocation across the four VM types to track the sudden changes in the arrival rate, with each type contributing differently. For instance, after an initial allocation of about 18 Large VMs during the first 40 minutes, their number drops sharply and even reaches zero. In contrast, the autoscaler relies more heavily on Medium VMs, whose processing capacity is closer to the predicted demand

and offers more flexibility in bucket selection; similar reasoning applies to the Small VMs.

Another metric considered for analyzing the autoscaler behavior is the utilization of the allocated VMs. Figure 8.8 shows the average utilization measured over 437 time slots of 15 seconds each. The figure shows that the utilization

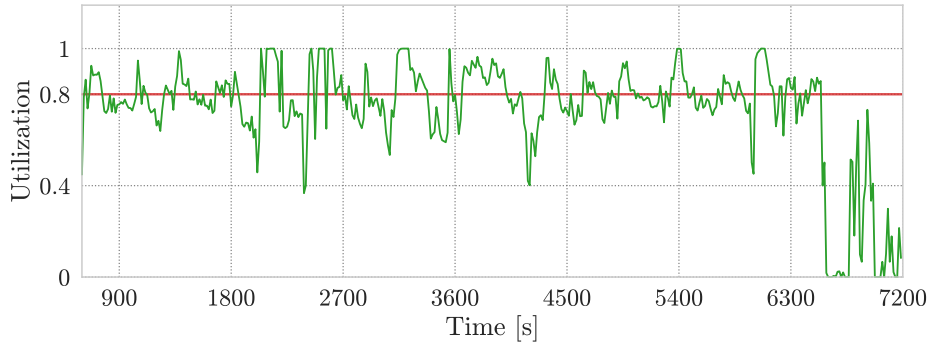


FIGURE 8.8: Average utilization of the allocated VMs per 15 second interval on the Redset workload. The red line refers to the desired utilization level of 0.8.

varies with time. Overall, the average utilization is 0.74, slightly below the target of 0.8. The autoscaler tends to over-provision in 256 slots, where the utilization drops to an average of 0.63, while in the remaining 181 slots it under-provisions and the utilization exceeds the target, reaching an average of 0.89.

8.2.5 COMPARISON OF HETEROGENEOUS AND HOMOGENEOUS RESOURCES

Finally, to assess the benefit of leveraging heterogeneous VMs with respect to a single VM type, the framework is evaluated on the Redset workload under four homogeneous configurations, each restricted to one of the four VM types. Table 8.8 summarizes the performance of the autoscaler for these configurations, and for the heterogeneous scenario for comparison. As expected, the performance of the homogeneous scenarios depends on the VM type. The performance, in terms of number of violations and monetary cost, achieved by the Small and Medium VM types is worse than that of the VM mix. For the two remaining scenarios, the performance is better in terms of cost and violations, respectively. The heterogeneous scenario is nonetheless preferable as it provides a good performance tradeoff and allows the autoscaler to automati-

Scenario	Type	Avg. utilization	Number of violations	Cost [USD]
Homogeneous	Small	0.73	196	12.25
	Medium	0.74	195	11.16
	Large	0.72	184	8.98
	XLarge	0.72	173	10.39
Heterogeneous	—	0.74	181	10.40

TABLE 8.8: Heterogeneous against single-type configurations on the Redset workload.

cally allocate the most appropriate mix of VMs without requiring Cloud users to manually identify the "best" VM type for their workloads.

CHAPTER 9

CONCLUSION AND FUTURE WORK

Resource management in Cloud and Cloud-Edge environments is a critical problem as resource allocation directly affects workload performance and cost. This thesis addressed two research problems in this domain: workflow scheduling and predictive autoscaling.

In the context of workflow scheduling, to minimize the execution time of workflows and the monetary cost of the resources, a probabilistic multi-objective optimization problem was formulated. The novel aspect of this work is the joint scheduling of multiple workflows, whose scheduling plans allow a better exploitation of the resources by interleaving the executions of tasks belonging to different workflows.

As the number of possible solutions makes the exact optimization intractable, the problem was solved by using a customized GA. In particular, the crossover and mutation operators used by the GA have been modified to be task dependency-aware. In addition, the strategies proposed for choosing the initial population allowed the GA to provide a wider range of scheduling plans, which could also favor high-priority workflows. To account for the performance variability of the underlying infrastructure, the fitness of each scheduling plan was evaluated through a Monte Carlo simulation.

The experiments have shown that the proposed approach produces scheduling plans that effectively balance execution time and monetary cost for multiple workflows across the Cloud-Edge Continuum. The Continuum is central to this

result: by exploiting both Cloud and Edge resources, the GA identifies hybrid scheduling plans that are not identified with Cloud-only or Edge-only infrastructures.

Results also showed that interleaving tasks belonging to different workflows, rather than running each workflow sequentially, makes better use of the available resources. The benefit persists as the workload grows in terms of the number of workflows, with execution times remaining comparable even though processing demand and data volume increase, indicating that interleaving tasks of different workflows make it possible to execute the additional load without significant slowdowns.

The probabilistic formulation revealed that performance uncertainty substantially shifts the resulting Pareto front towards scheduling plans with longer execution times and higher cost with respect to a deterministic scenario.

A similar effect was observed by associating deadline constraints to individual workflows where, under sufficiently high variability and tight deadlines, the GA fails to identify any feasible scheduling plan.

In the context of autoscaling, correctly exploiting the elasticity offered by Cloud environments can provide significant benefits in terms of performance and leasing costs. However, since provisioned VMs require a boot time, provisioning choices must anticipate the demand rather than react to it.

To this end, a Load-Aware Predictive autoscaling policy that anticipates workload changes has been proposed. At each execution of the policy, the predictions of the future workload demand are combined with estimates of the backlog of unprocessed requests to determine the VMs able to keep the overall resource utilization at the desired level. By design, scale-out actions are triggered promptly to meet rising demand, whereas to avoid the performance degradation caused by premature deallocations, scale-in actions are triggered only after persistent over-provisioning.

The proposed policy was implemented as a component of a simulation environment built on the CloudSim simulation toolkit.

The experiments have shown that the Load-Aware Predictive policy regulates the allocated resources to timely match the workload demands in infrastructures with both homogeneous and heterogeneous VM types. In particular the policy outperformed the reactive baseline, reducing leasing costs without any loss in the responsiveness experienced by the requests.

The importance of the choice of the desired utilization level was also observed. As expected, high values cause resource saturation and performance

degradation, while low values lead to increased leasing times without significant performance benefits.

The results presented in this thesis are subject to some limitations. Both frameworks were assessed through simulation, and while the workloads driving the experiments include real scientific workflows and arrival patterns drawn from operational systems, the underlying infrastructure has been modeled by treating processing capacities, bandwidths, and leasing costs as configuration parameters rather than measured values.

Resource management was addressed from two perspectives that complement each other: the allocation of resources and the scheduling of a batch workload modeled as workflows, and the allocation and deallocation of resources for the processing of an interactive workload.

As future research directions, several extensions of the resource management problems studied in this thesis are envisioned.

Machine learning techniques can be used for designing heuristics that learn from past executions, both for guiding scheduling decisions on historical data in the context of workflow scheduling and for forecasting incoming workloads in the context of autoscaling.

In addition, more sophisticated workload models able to capture the complex structure of modern applications, whose dependency graphs are not always known in advance, could be developed.

Constraints related to data privacy and confidentiality could also be introduced, where certain tasks of a workflow or certain workflows may be required to be executed on specific resources to keep sensitive data within a trusted domain.

ACRONYMS

ACO	Ant Colony Optimization
DAG	Directed Acyclic Graph
DVFS	Dynamic Voltage and Frequency Scaling
GA	Genetic Algorithm
HEFT	Heterogeneous Earliest Finish Time
IoT	Internet of Things
KHPA	Kubernetes Horizontal Pod Autoscaler
MAPE	Monitor–Analyze–Plan–Execute
MAPE-K	Monitor–Analyze–Plan–Execute over a shared Knowledge
MI	Millions of Instructions
MIPS	Million Instructions Per Second
MOHEFT	Multi-Objective Heterogeneous Earliest Finish Time
NSGA-II	Non-dominated Sorting Genetic Algorithm II
PSO	Particle Swarm Optimization
QoS	Quality of Service
SLA	Service Level Agreement
VM	Virtual Machine

LIST OF PUBLICATIONS

L. Zanussi, D. Tessera, L. Massari, and M. C. Calzarossa. Workflow Scheduling in the Cloud-Edge Continuum. In L. Barolli, editor, *Proc. of the International Conference on Advanced Information Networking and Applications (AINA)*, volume 203 of *Lecture Notes on Data Engineering and Communications Technologies*, pages 182–190. Springer, 2024.

A. Esposito, R. Aversa, E. Barbierato, M. C. Calzarossa, B. Di Martino, L. Massari, I. G. Mongiardo, D. Tessera, S. Venticinque, L. Zanussi, and R. Zieni. Methodologies for the Parallelization, Performance Evaluation and Scheduling of Applications for the Cloud-Edge Continuum. In L. Barolli, editor, *Proc. of the International Conference on Advanced Information Networking and Applications (AINA)*, volume 203 of *Lecture Notes on Data Engineering and Communications Technologies*, pages 254–263. Springer, 2024.

L. Zanussi, D. Tessera, L. Massari, B. Bermejo, C. Juiz, and M. C. Calzarossa. Auto-scaling in Scenarios with Heterogeneous Cloud Resources. In *Proc. of the Companion of the 17th ACM/SPEC International Conference on Performance Engineering*, ICPE Companion '26, pages 213–217. ACM, 2026.

L. Zanussi, D. Tessera, L. Massari, B. Bermejo, C. Juiz, and M. C. Calzarossa. Load-aware predictive auto-scaling framework for cloud environments. *Cluster Computing*, 29(3):197, 2026.

L. Zanussi, D. Tessera, L. Massari, and M. C. Calzarossa. Scheduling multiple workflows in Cloud-Edge Environments. *Journal of Network and Systems Management*, 2026. (submitted: under second revision).

A. Esposito, M. Calzarossa, P. Dazzi, B. Di Martino, A. Longo, L. Marcellino, L. Massari, R. Montella, D. Tessera, and L. Zanussi. RESCUE-IT: a dual-mode agentic AI framework for resource-efficient emergency response over the cloud-edge continuum. In L. Barolli, F. Hussain, and F. Amato, editors, *Proc. of the Advances on P2P, Parallel, Grid, Cloud and Internet Computing*, Lecture Notes on Data Engineering and Communications Technologies. Springer, 2026. (accepted for publication).

BIBLIOGRAPHY

- [1] D. Smith. Cloud Computing in 2028: From Technology to Business Necessity. Technical report, Gartner Research, 2023.
- [2] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli. Fog computing and its role in the internet of things. In *Proc. of the 1st MCC Workshop on Mobile Cloud Computing*, pages 13–16. ACM, 2012.
- [3] J. D. Ullman. NP-complete scheduling problems. *Journal of Computer and System Sciences*, 10(3):384–393, 1975.
- [4] 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*, 13(3):260–274, 2002.
- [5] J. J. Durillo, H. M. Fard, and R. Prodan. MOHEFT: A multi-objective list-based method for workflow scheduling. In *Proc. of the 4th International Conference on Cloud Computing Technology and Science Proceedings*, pages 185–192. IEEE, 2012.
- [6] J. H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. The MIT Press, 1992.
- [7] J. Kennedy and R. Eberhart. Particle swarm optimization. In *Proc. of the International Conference on Neural Networks (ICNN’95)*, volume 4, pages 1942–1948, 1995.
- [8] C. Coello, G. Pulido, and M. Lechuga. Handling multiple objectives with particle swarm optimization. *IEEE Transactions on Evolutionary Computation*, 8(3):256–279, 2004.

- [9] M. Dorigo and G. Di Caro. Ant colony optimization: a new meta-heuristic. In *Proc. of the 1999 Congress on Evolutionary Computation (CEC'99)*, volume 2, pages 1470–1477, 1999.
- [10] X.S. Yang. Firefly algorithms for multimodal optimization. In *Proc. of the International symposium on stochastic algorithms*, pages 169–178. Springer, 2009.
- [11] D. Karaboga. An Idea Based on Honey Bee Swarm for Numerical Optimization. Technical report TR06, Erciyes University, Engineering Faculty, Computer Engineering Department, 2005.
URL: https://abc.erciyes.edu.tr/pub/tr06_2005.pdf
- [12] S. Mirjalili, A. H. Gandomi, S. Z. Mirjalili, S. Saremi, H. Faris, et al. Salp Swarm Algorithm: a bio-inspired optimizer for engineering design problems. *Advances in Engineering Software*, 114:163–191, 2017.
- [13] J. Kephart and D. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.
- [14] P. Arcaini, E. Riccobene, and P. Scandurra. Modeling and Analyzing MAPE-K Feedback Loops for Self-Adaptation. In *Proc. of the 10th IEEE/ACM International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 13–23, 2015.
- [15] N. Khaledian, M. Voelp, S. Azizi, and M. H. Shirvani. AI-based & heuristic workflow scheduling in cloud and fog computing: a systematic review. *Cluster Computing*, 27(8):10265–10298, 2024.
- [16] 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*, 66:64–82, 2016.
- [17] AR. Arunarani, D. Manjula, and V. Sugumaran. Task scheduling techniques in cloud computing: A literature survey. *Future Generation Computer Systems*, 91:407–415, 2019.
- [18] M. Adhikari, T. Amgoth, and S. N. Srirama. A Survey on Scheduling Strategies for Workflows in Cloud Environment and Emerging Trends. *ACM Computing Surveys*, 52(4):1–36, 2020.

- [19] M. Hosseinzadeh, M. Y. Ghafour, H. K. Hama, B. Vo, and A. Khoshnevis. Multi-Objective Task and Workflow Scheduling Approaches in Cloud Computing: a Comprehensive Review. *Journal of Grid Computing*, 18(3):327–356, 2020.
- [20] E. Saeedizade and M. Ashtiani. Scientific workflow scheduling algorithms in cloud environments: a comprehensive taxonomy, survey, and future directions. *Journal of Scheduling*, 28(1):1–63, 2025.
- [21] M. A. Rodriguez and R. Buyya. Deadline Based Resource Provisioning and Scheduling Algorithm for Scientific Workflows on Clouds. *IEEE Transactions on Cloud Computing*, 2(2):222–235, 2014.
- [22] A. Verma and S. Kaushal. Deadline constraint heuristic-based genetic algorithm for workflow scheduling in cloud. *International Journal of Grid and Utility Computing*, 5(2):96, 2014.
- [23] A. M. Chirkin, A. S. Z. Belloum, S. V. Kovalchuk, M. X. Makkes, M. A. Melnik, et al. Execution time estimation for workflow scheduling. *Future Generation Computer Systems*, 75:376–387, 2017.
- [24] L. Liu, M. Zhang, R. Buyya, and Q. Fan. Deadline-constrained coevolutionary genetic algorithm for scientific workflow scheduling in cloud computing. *Concurrency and Computation: Practice and Experience*, 29(5):e3942, 2017.
- [25] Q. Wu, F. Ishikawa, Q. Zhu, Y. Xia, and J. Wen. Deadline-Constrained Cost Optimization Approaches for Workflow Scheduling in Clouds. *IEEE Transactions on Parallel and Distributed Systems*, 28(12):3401–3412, 2017.
- [26] A. Iranmanesh and H. R. Naji. DCHG-TS: a deadline-constrained and cost-effective hybrid genetic algorithm for scientific workflow scheduling in cloud computing. *Cluster Computing*, 24(2):667–681, 2021.
- [27] Z. Zhu, G. Zhang, M. Li, and X. Liu. Evolutionary Multi-Objective Workflow Scheduling in Cloud. *IEEE Transactions on Parallel and Distributed Systems*, 27(5):1344–1357, 2016.
- [28] H. Aziza and S. Krichen. A hybrid genetic algorithm for scientific workflow scheduling in cloud environment. *Neural Computing and Applications*, 32(18):15263–15278, 2020.

- [29] X. Xia, H. Qiu, X. Xu, and Y. Zhang. Multi-objective workflow scheduling based on genetic algorithm in cloud environment. *Information Sciences*, 606:38–59, 2022.
- [30] Z.G. Chen, Z.H. Zhan, Y. Lin, Y.J. Gong, T.L. Gu, et al. Multiobjective Cloud Workflow Scheduling: A Multiple Populations Ant Colony System Approach. *IEEE Transactions on Cybernetics*, 49(8):2912–2926, 2019.
- [31] A. Belgacem and K. Beghdad-Bey. Multi-objective workflow scheduling in cloud computing: trade-off between makespan and cost. *Cluster Computing*, 25(1):579–595, 2022.
- [32] X. Wang, C. S. Yeo, R. Buyya, and J. Su. Optimizing the makespan and reliability for workflow applications with reputation and a look-ahead genetic algorithm. *Future Generation Computer Systems*, 27(8):1124–1134, 2011.
- [33] A. Verma and S. Kaushal. A hybrid multi-objective Particle Swarm Optimization for scientific workflow scheduling. *Parallel Computing*, 62:1–19, 2017.
- [34] M. Zeedan, G. Attiya, and N. El-Fishawy. Enhanced hybrid multi-objective workflow scheduling approach based artificial bee colony in cloud computing. *Computing*, 105(1):217–247, 2023.
- [35] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.
- [36] J. Zhou, T. Wang, P. Cong, P. Lu, T. Wei, et al. Cost and makespan-aware workflow scheduling in hybrid clouds. *Journal of Systems Architecture*, 100:101631, 2019.
- [37] X.Q. Pham, N. D. Man, N. D. T. Tri, N. Q. Thai, and E.N. Huh. A cost-and performance-effective approach for task scheduling based on collaboration between cloud and fog computing. *International Journal of Distributed Sensor Networks*, 13(11), 2017.
- [38] S. Ijaz, E. U. Munir, S. G. Ahmad, M. M. Rafique, and O. F. Rana. Energy-makespan optimization of workflow scheduling in fog-cloud computing. *Computing*, 103(9):2033–2059, 2021.

- [39] G. Singh and A. K. Chaturvedi. Hybrid modified particle swarm optimization with genetic algorithm (GA) based workflow scheduling in cloud-fog environment for multi-objective optimization. *Cluster Computing*, 27(2):1947–1964, 2024.
- [40] Y. Xie, Y. Zhu, Y. Wang, Y. Cheng, R. Xu, et al. A novel directional and non-local-convergent particle swarm optimization based workflow scheduling in cloud-edge environment. *Future Generation Computer Systems*, 97:361–378, 2019.
- [41] N. Bacanin, M. Zivkovic, T. Bezdan, K. Venkatachalam, and M. Abouhawwash. Modified firefly algorithm for workflow scheduling in cloud-edge environment. *Neural Computing and Applications*, 34(11):9043–9068, 2022.
- [42] D. Alsadie and M. Alsulami. Enhancing workflow efficiency with a modified Firefly Algorithm for hybrid cloud edge environments. *Scientific Reports*, 14(1):24675, 2024.
- [43] M. Ghobaei-Arani, A. Souri, and A. A. Rahmanian. Resource Management Approaches in Fog Computing: a Comprehensive Review. *Journal of Grid Computing*, 18(1):1–42, 2020.
- [44] C. Guerrero, I. Lera, and C. Juiz. Genetic-based optimization in fog computing: Current trends and research opportunities. *Swarm and Evolutionary Computation*, 72:101094, 2022.
- [45] Y. Sun, F. Lin, and H. Xu. Multi-objective Optimization of Resource Scheduling in Fog Computing Using an Improved NSGA-II. *Wireless Personal Communications*, 102(2):1369–1385, 2018.
- [46] R. O. Aburukba, M. AliKarrar, T. Landolsi, and K. El-Fakih. Scheduling Internet of Things requests to minimize latency in hybrid Fog-Cloud computing. *Future Generation Computer Systems*, 111:539–551, 2020.
- [47] R. O. Aburukba, T. Landolsi, and D. Omer. A heuristic scheduling approach for fog-cloud computing environment with stationary IoT devices. *Journal of Network and Computer Applications*, 180:102994, 2021.

- [48] P. Memari, S. S. Mohammadi, F. Jolai, and R. Tavakkoli-Moghaddam. A latency-aware task scheduling algorithm for allocating virtual machines in a cost-effective and time-sensitive fog-cloud architecture. *The Journal of Supercomputing*, 78(1):93–122, 2022.
- [49] S. Kabirzadeh, D. Rahbari, and M. Nickray. A hyper heuristic algorithm for scheduling of fog networks. In *Proc. of the 21st Conference of Open Innovations Association (FRUCT)*, pages 148–155, 2017.
- [50] F. A. Saif, R. Latip, Z. M. Hanapi, and K. Shafinah. Multi-Objective Grey Wolf Optimizer Algorithm for Task Scheduling in Cloud-Fog Computing. *IEEE Access*, 11:20635–20646, 2023.
- [51] A. S. Abohamama, A. El-Ghamry, and E. Hamouda. Real-Time Task Scheduling Algorithm for IoT-Based Applications in the Cloud–Fog Environment. *Journal of Network and Systems Management*, 30(4), 2022.
- [52] A. Najafizadeh, A. Salajegheh, A. M. Rahmani, and A. Sahafi. Multi-objective Task Scheduling in cloud-fog computing using goal programming approach. *Cluster Computing*, 25(1):141–165, 2022.
- [53] S. M. Hussain and G. R. Begh. Hybrid heuristic algorithm for cost-efficient QoS aware task scheduling in fog–cloud environment. *Journal of Computational Science*, 64:101828, 2022.
- [54] G. Agarwal, S. Gupta, R. Ahuja, and A. K. Rai. Multiprocessor task scheduling using multi-objective hybrid genetic Algorithm in Fog–cloud computing. *Knowledge-Based Systems*, 272:110563, 2023.
- [55] M. Goudarzi, M. Palaniswami, and R. Buyya. Scheduling IoT Applications in Edge and Fog Computing Environments: A Taxonomy and Future Directions. *ACM Computing Surveys*, 55(7), 2022.
- [56] R. Bouabdallah and F. Fakhfakh. Workflow Scheduling in Cloud–Fog Computing Environments: A Systematic Literature Review. *Concurrency and Computation: Practice and Experience*, 36(28):e8304, 2024.
- [57] Q. Jiang, Y. C. Lee, and A. Y. Zomaya. Executing Large Scale Scientific Workflow Ensembles in Public Clouds. In *Proc. of the 44th International Conference on Parallel Processing*, pages 520–529. IEEE, 2015.

- [58] N. Shafinezhad, H. Abrishami, B. Zolfaghari, S. Abrishami, and A. Morvaridi. Workflow ensemble scheduling in IaaS cloud: a gap analysis perspective under deadline and budget constraints. *The Journal of Supercomputing*, 81(11):1168, 2025.
- [59] M. H. Hilman, M. A. Rodriguez, and R. Buyya. Multiple Workflows Scheduling in Multi-tenant Distributed Systems: A Taxonomy and Future Directions. *ACM Computing Surveys*, 53(1):1–39, 2021.
- [60] H. Zhao and R. Sakellariou. Scheduling multiple DAGs onto heterogeneous systems. In *Proc. of the 20th IEEE International Parallel & Distributed Processing Symposium*, 2006.
- [61] L. F. Bittencourt and E. R. M. Madeira. Towards the Scheduling of Multiple Workflows on Computational Grids. *Journal of Grid Computing*, 8(3):419–441, 2010.
- [62] K. Bochenina, N. Butakov, and A. Boukhanovsky. Static scheduling of multiple workflows with soft deadlines in non-dedicated heterogeneous environments. *Future Generation Computer Systems*, 55:51–61, 2016.
- [63] M. Malawski, G. Juve, E. Deelman, and J. Nabrzyski. Cost- and deadline-constrained provisioning for scientific workflow ensembles in IaaS clouds. In *Proc. of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012.
- [64] M. Malawski, G. Juve, E. Deelman, and J. Nabrzyski. Algorithms for cost- and deadline-constrained provisioning for scientific workflow ensembles in IaaS clouds. *Future Generation Computer Systems*, 48:1–18, 2015.
- [65] T. A. L. Genez, L. F. Bittencourt, R. Sakellariou, and E. R. M. Madeira. A flexible scheduler for workflow ensembles. In *Proc. of the 9th International Conference on Utility and Cloud Computing*, pages 55–62. ACM, 2016.
- [66] B. Zolfaghari and S. Abrishami. A multi-class workflow ensemble management system using on-demand and spot instances in cloud. *Future Generation Computer Systems*, 137:97–110, 2022.
- [67] L. Chen, G. Liu, J. Zhang, and X. Zhang. Scheduling ensemble workflows on hybrid resources in IaaS clouds. *Computing*, 107(1):22, 2024.

- [68] N. Shafinezhad, H. Abrishami, and S. Abrishami. An Adaptive Budget and Deadline-aware Algorithm for Scheduling Workflows Ensemble in IaaS Clouds. In *Proc. of the 13th International Conference on Computer and Knowledge Engineering (ICCKE)*, pages 432–438, 2023.
- [69] H. Siar and M. Izadi. Offloading Coalition Formation for Scheduling Scientific Workflow Ensembles in Fog Environments. *Journal of Grid Computing*, 19(3):34, 2021.
- [70] Q.V. Pham, F. Fang, V. N. Ha, M. J. Piran, M. Le, et al. A Survey of Multi-Access Edge Computing in 5G and Beyond: Fundamentals, Technology Integration, and State-of-the-Art. *IEEE Access*, 8:116974–117017, 2020.
- [71] H. M. D. Kabir, A. Khosravi, S. K. Mondal, M. Rahman, S. Nahavandi, et al. Uncertainty-aware Decisions in Cloud Computing: Foundations and Future Directions. *ACM Computing Surveys*, 54(4):74:1–74:30, 2021.
- [72] A. Tchernykh, U. Schwiegelsohn, V. Alexandrov, and E.g. Talbi. Towards Understanding Uncertainty in Cloud Computing Resource Provisioning. *Procedia Computer Science*, 51:1772–1781, 2015.
- [73] A. Marotta, F. D’Andreagiovanni, A. Kassler, and E. Zola. On the energy cost of robustness for green virtual network function placement in 5G virtualized infrastructures. *Computer Networks*, 125:64–75, 2017.
- [74] B. Farkiani, B. Bakhshi, and S. Ali MirHassani. Stochastic virtual network embedding via accelerated Benders decomposition. *Future Generation Computer Systems*, 94:199–213, 2019.
- [75] S. Chaisiri, B.S. Lee, and D. Niyato. Optimization of Resource Provisioning Cost in Cloud Computing. *IEEE Transactions on Services Computing*, 5(2):164–177, 2012.
- [76] J. Meena, M. Kumar, and M. Vardhan. Cost Effective Genetic Algorithm for Workflow Scheduling in Cloud Under Deadline Constraint. *IEEE Access*, 4:5065–5082, 2016.
- [77] M. L. Della Vedova, D. Tessaera, and M. C. Calzarossa. Probabilistic provisioning and scheduling in uncertain Cloud environments. In *Proc. of the 2016 IEEE Symposium on Computers and Communication (ISCC)*, pages 797–803. IEEE, 2016.

- [78] H. M. Fard, S. Ristov, and R. Prodan. Handling the uncertainty in resource performance for executing workflow applications in clouds. In *Proc. of the 9th International Conference on Utility and Cloud Computing*, pages 89–98. ACM, 2016.
- [79] T. A. L. Genez, L. F. Bittencourt, R. Sakellariou, and E. R. M. Madeira. A Robust Scheduler for Workflow Ensembles under Uncertainties of Available Bandwidth. In *Proc. of the 10th International Conference on Cloud Computing (CLOUD)*, pages 504–511. IEEE, 2017.
- [80] H. Chen, X. Zhu, G. Liu, and W. Pedrycz. Uncertainty-Aware Online Scheduling for Real-Time Workflows in Cloud Service Environment. *IEEE Transactions on Services Computing*, 14(4):1167–1178, 2021.
- [81] J. Liu, J. Ren, W. Dai, D. Zhang, P. Zhou, et al. Online Multi-Workflow Scheduling under Uncertain Task Execution Time in IaaS Clouds. *IEEE Transactions on Cloud Computing*, 9(3):1180–1194, 2021.
- [82] M. C. Calzarossa, M. L. Della Vedova, and D. Tessa. A methodological framework for cloud resource provisioning and scheduling of data parallel applications under uncertainty. *Future Generation Computer Systems*, 93:212–223, 2019.
- [83] M. C. Calzarossa, L. Massari, G. Nebbione, M. L. Della Vedova, and D. Tessa. Tuning Genetic Algorithms for Resource Provisioning and Scheduling in Uncertain Cloud Environments: Challenges and Findings. In *Proc. of the 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 174–180, 2019.
- [84] G. L. Stavrinos and H. D. Karatza. A hybrid approach to scheduling real-time IoT workflows in fog and cloud environments. *Multimedia Tools and Applications*, 78(17):24639–24655, 2019.
- [85] M. C. Calzarossa, M. L. Della Vedova, L. Massari, G. Nebbione, and D. Tessa. Multi-Objective Optimization of Deadline and Budget-Aware Workflow Scheduling in Uncertain Clouds. *IEEE Access*, 9:89891–89905, 2021.
- [86] V. De Maio and D. Kimovski. Multi-objective scheduling of extreme data scientific workflows in Fog. *Future Generation Computer Systems*, 106:171–184, 2020.

- [87] N. Rizvi, D. Ramesh, P. C. S. Rao, and K. Mondal. Intelligent Salp Swarm Scheduler With Fitness Based Quasi-Reflection Method for Scientific Workflows in Hybrid Cloud-Fog Environment. *IEEE Transactions on Automation Science and Engineering*, 20(2):862–877, 2023.
- [88] M. Mokni, S. Yassa, J. E. Hajlaoui, M. N. Omri, and R. Chelouah. Multi-objective fuzzy approach to scheduling and offloading workflow tasks in Fog–Cloud computing. *Simulation Modelling Practice and Theory*, 123:102687, 2023.
- [89] E. H. Mamdani and S. Assilian. An experiment in linguistic synthesis with a fuzzy logic controller. *International Journal of Man-Machine Studies*, 7(1):1–13, 1975.
- [90] C. Qu, R. N. Calheiros, and R. Buyya. Auto-Scaling Web Applications in Clouds: A Taxonomy and Survey. *ACM Computing Surveys*, 51(4), 2019.
- [91] S. Verma and A. Bala. Auto-scaling techniques for IoT-based cloud applications: a review. *Cluster Computing*, 24(3):2425–2459, 2021.
- [92] J. Dogani, R. Namvar, and F. Khunjush. Auto-scaling techniques in container-based cloud and edge/fog computing: Taxonomy and survey. *Computer Communications*, 209:120–150, 2023.
- [93] B. Jeong and Y.S. Jeong. Autoscaling techniques in cloud-native computing: A comprehensive survey. *Computer Science Review*, 58:100791, 2025.
- [94] T. Lorigo-Botran, J. Miguel-Alonso, and J. A. Lozano. A Review of Auto-scaling Techniques for Elastic Applications in Cloud Environments. *Journal of Grid Computing*, 12(4):559–592, 2014.
- [95] A. Barnawi, S. Sakr, W. Xiao, and A. Al-Barakati. The views, measurements and challenges of elasticity in the cloud: A review. *Computer Communications*, 154:111–117, 2020.
- [96] M. Catillo, U. Villano, and M. Rak. A survey on auto-scaling: how to exploit cloud elasticity. *International Journal of Grid and Utility Computing*, 14(1):37–50, 2023.

- [97] M. A. Netto, C. Cardonha, R. L. Cunha, and M. D. Assuncao. Evaluating Auto-scaling Strategies for Cloud Computing Environments. In *Proc. of the 22nd International Symposium on Modelling, Analysis & Simulation of Computer and Telecommunication Systems*, pages 187–196. IEEE, 2014.
- [98] W. Iqbal, M. N. Dailey, and D. Carrera. Low Cost Quality Aware Multi-tier Application Hosting on the Amazon Cloud. In *Proc. of the 2nd International Conference on Future Internet of Things and Cloud*, pages 202–209, 2014.
- [99] D. R. Augustyn and L. Warchal. Metrics-Based Auto Scaling Module for Amazon Web Services Cloud Platform. In S. Kozielski, D. Mrozek, P. Kasprowski, B. Małysiak-Mrozek, and D. Kostrzewa, editors, *Beyond Databases, Architectures, and Structures*. Volume 716, Communications in Computer and Information Science, pages 42–52. Springer, 2017.
- [100] M. C. Calzarossa, L. Massari, and D. Tessera. Evaluation of cloud autoscaling strategies under different incoming workload patterns. *Concurrency and Computation: Practice and Experience*, 32(17):e5667, 2020.
- [101] A. Gandhi, M. Harchol-Balter, R. Raghunathan, and M. A. Kozuch. AutoScale: Dynamic, Robust Capacity Management for Multi-Tier Data Centers. *ACM Transactions on Computer Systems*, 30(4):14:1–14:26, 2012.
- [102] S. Heidari and R. Buyya. A Cost-Efficient Auto-Scaling Algorithm for Large-Scale Graph Processing in Cloud Environments with Heterogeneous Resources. *IEEE Transactions on Software Engineering*, 47(8):1729–1741, 2021.
- [103] T. Tournaire, H. Castel-Taleb, and E. Hyon. Efficient Computation of Optimal Thresholds in Cloud Auto-scaling Systems. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*, 8(4):1–31, 2023.
- [104] E. Casalicchio and V. Perciballi. Auto-Scaling of Containers: The Impact of Relative and Absolute Metrics. In *Proc. of the 2nd International Workshops on Foundations and Applications of Self* Systems (FAS*W)*, pages 207–214, 2017.

- [105] E. Casalicchio. A study on performance measures for auto-scaling CPU-intensive containerized applications. *Cluster Computing*, 22(3):995–1006, 2019.
- [106] M. Straesser, S. Geissler, S. Lange, L. K. Schumann, T. Hossfeld, et al. Trust your local scaler: A continuous, decentralized approach to autoscaling. *Performance Evaluation*, 167:102452, 2025.
- [107] B. Liu, R. Buyya, and A. Nadjaran Toosi. A Fuzzy-Based Auto-scaler for Web Applications in Cloud Computing Environments. In C. Pahl, M. Vukovic, J. Yin, and Q. Yu, editors, *Proc. of the 16th International Conference on Service-Oriented Computing*, volume 11236 of *Lecture Notes in Computer Science*, pages 797–811. Springer, 2018.
- [108] G. Quattrocchi, E. Incerto, R. Pincioli, C. Trubiani, and L. Baresi. Autoscaling Solutions for Cloud Applications Under Dynamic Workloads. *IEEE Transactions on Services Computing*, 17(3):804–820, 2024.
- [109] G. Russo Russo, V. Cardellini, and F. Lo Presti. Hierarchical Auto-scaling Policies for Data Stream Processing on Heterogeneous Resources. *ACM Transactions on Autonomous and Adaptive Systems*, 18(4):14:1–14:44, 2023.
- [110] X. Feng, S. Zhang, T. Jiao, C. Guo, and J. Song. Adaptive container auto-scaling for fluctuating workloads in cloud. *Future Generation Computer Systems*, 172:107872, 2025.
- [111] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes. Borg, Omega, and Kubernetes. Lessons learned from three container-management systems over a decade. *Communications of the ACM*, 59(5):50–57, 2016.
- [112] D. Bernstein. Containers and Cloud: From LXC to Docker to Kubernetes. *IEEE Cloud Computing*, 1(3):81–84, 2014.
- [113] W. Iqbal, A. Erradi, and A. Mahmood. Dynamic workload patterns prediction for proactive auto-scaling of web applications. *Journal of Network and Computer Applications*, 124:94–107, 2018.
- [114] M. C. Calzarossa, L. Massari, and D. Tessera. Workload Characterization: A Survey Revisited. *ACM Computing Surveys*, 48(3), 2016.

- [115] M. C. Calzarossa, M. L. Della Vedova, L. Massari, D. Petcu, M. I. M. Tabash, et al. Workloads in the Clouds. In L. Fiondella and A. Puliafito, editors, *Principles of Performance and Reliability Modeling and Evaluation*, pages 525–550. Springer, 2016.
- [116] M. Masdari and A. Khoshnevis. A survey and classification of the workload forecasting methods in cloud computing. *Cluster Computing*, 23(4):2399–2424, 2020.
- [117] E. Radhika and G. Sudha Sadasivam. A review on prediction based autoscaling techniques for heterogeneous applications in cloud environment. *Materials Today: Proceedings*, 45:2793–2800, 2021.
- [118] M. C. Calzarossa, M. L. D. Vedova, L. Massari, G. Nebbione, and D. Tessera. Modeling and predicting dynamics of heterogeneous workloads for cloud environments. In *Proc. of the IEEE Symposium on Computers and Communications (ISCC)*. IEEE, 2019.
- [119] N. Roy, A. Dubey, and A. Gokhale. Efficient Autoscaling in the Cloud Using Predictive Models for Workload Forecasting. In *Proc. of the 4th International Conference on Cloud Computing (CLOUD)*, pages 500–507. IEEE, 2011.
- [120] Z. Gong, X. Gu, and J. Wilkes. PRESS: PRedictive Elastic ReSource Scaling for cloud systems. In *Proc. of the 6th International Conference on Network and Service Management*, pages 9–16. IEEE, 2010.
- [121] J. Kumar and A. K. Singh. Workload prediction in cloud using artificial neural network and adaptive differential evolution. *Future Generation Computer Systems*, 81:41–52, 2018.
- [122] F. Tahir, M. Abdullah, F. Bukhari, K. M. Almustafa, and W. Iqbal. On-line Workload Burst Detection for Efficient Predictive Autoscaling of Applications. *IEEE Access*, 8:73730–73745, 2020.
- [123] S. Verma and A. Bala. Efficient Auto-scaling for Host Load Prediction through VM migration in Cloud. *Concurrency and Computation: Practice and Experience*, 36(4):e7925, 2024.
- [124] A. Y. Nikraves, S. A. Ajila, and C.H. Lung. An autonomic prediction suite for cloud resource provisioning. *Journal of Cloud Computing*, 6(1):3, 2017.

- [125] W. Iqbal, A. Erradi, M. Abdullah, and A. Mahmood. Predictive Auto-Scaling of Multi-Tier Applications Using Performance Varying Cloud Resources. *IEEE Transactions on Cloud Computing*, 10(1):595–607, 2022.
- [126] D. Zhou, H. Chen, K. Shang, G. Cheng, J. Zhang, et al. *Cushion* : A proactive resource provisioning method to mitigate SLO violations for containerized microservices. *IET Communications*, 16(17):2105–2122, 2022.
- [127] Y. Alidoost Alanagh, M. Firouzi, A. Rasouli Kenari, and M. Shamsi. Introducing an adaptive model for auto-scaling cloud computing based on workload classification. *Concurrency and Computation: Practice and Experience*, 35(22):e7720, 2023.
- [128] M. Abdullah, W. Iqbal, J. L. Berral, J. Polo, and D. Carrera. Burst-Aware Predictive Autoscaling for Containerized Microservices. *IEEE Transactions on Services Computing*, 15(3):1448–1460, 2022.
- [129] S. Chouliaras and S. Sotiriadis. An adaptive auto-scaling framework for cloud resource provisioning. *Future Generation Computer Systems*, 148:173–183, 2023.
- [130] J. Park, B. Jeong, J. Jeon, and Y.S. Jeong. Burst-Aware Horizontal Autoscaling Based on Deep Learning for Stable Microservices. *Human-centric Computing and Information Sciences*, 14:67, 2024.
- [131] Z. Wang, S. Zhu, J. Li, W. Jiang, K. K. Ramakrishnan, et al. DeepScaling: microservices autoscaling for stable CPU utilization in large scale cloud systems. In *Proc. of the 13th Symposium on Cloud Computing*, pages 16–30. ACM, 2022.
- [132] B. Kumar, A. Verma, P. Verma, and A. Bennour. Optimizing resource allocation in cloud-native applications through proactive autoscaling with the InformerAutoScale model. *The Journal of Supercomputing*, 81(9):1077, 2025.
- [133] Y. Zhang, H. Li, Y. Zeng, and Z. Wu. Predictive Auto Scaling and Cost Optimization Using Machine Learning in AWS Cloud Environments. In *Proc. of the 2nd International Symposium on Integrated Circuit Design and Integrated Systems*, pages 161–167. ACM, 2025.
- [134] D.D. Vu, M.N. Tran, and Y. Kim. Predictive Hybrid Autoscaling for Containerized Applications. *IEEE Access*, 10:109768–109778, 2022.

- [135] A. K. Mogal and V. P. Sonaje. Predictive Autoscaling for Containerized Applications Using Machine Learning. In *Proc. of the 1st International Conference on Cognitive, Green and Ubiquitous Computing (IC-CGU)*, pages 1–6. IEEE, 2024.
- [136] M. A. Razzaq, J. A. Mahar, M. Ahmad, N. Saher, A. Mehmood, et al. Hybrid Auto-Scaled Service-Cloud-Based Predictive Workload Modeling and Analysis for Smart Campus System. *IEEE Access*, 9:42081–42089, 2021.
- [137] A. Gandhi, P. Dube, A. Karve, A. Kochut, and L. Zhang. Adaptive, Model-driven Autoscaling for Cloud Applications. In *Proc. of the 11th International Conference on Autonomic Computing (ICAC '14)*, pages 57–64, 2014.
- [138] A. Gandhi, P. Dube, A. Karve, A. Kochut, and L. Zhang. Model-driven optimal resource scaling in cloud. *Software & Systems Modeling*, 17(2):509–526, 2018.
- [139] H. Qian, Q. Wen, L. Sun, J. Gu, Q. Niu, et al. RobustScaler: QoS-Aware Autoscaling for Complex Workloads. In *Proc. of the 38th IEEE International Conference on Data Engineering (ICDE)*, pages 2762–2775. IEEE, 2022.
- [140] Z. Pan, Y. Wang, Y. Zhang, S. B. Yang, Y. Cheng, et al. MagicScaler: Uncertainty-Aware, Predictive Autoscaling. *Proceedings of the VLDB Endowment*, 16(12):3808–3821, 2023.
- [141] H. Hang, X. Tang, J. Sun, L. Bao, D. Lo, et al. Robust Auto-Scaling with Probabilistic Workload Forecasting for Cloud Databases. In *Proc. of the 40th IEEE International Conference on Data Engineering (ICDE)*, pages 4016–4029, 2024.
- [142] M. Abdullah, R. Munir, W. Iqbal, S. A. Khan, S. U. R. Baig, et al. Cost and performance-effective dynamic VM type selection in auto-scaling using reinforcement learning. *Computing*, 107(9):177, 2025.
- [143] T. Theodoropoulos, Y. S. Patel, U. Zdun, P. Townend, I. Korontanis, et al. GraphOpticon: A Global proactive horizontal autoscaler for improved service performance & resource consumption. *Future Generation Computer Systems*, 174:107926, 2026.

- [144] J. Sahni and D. P. Vidyarthi. Heterogeneity-aware adaptive auto-scaling heuristic for improved QoS and resource usage in cloud environments. *Computing*, 99(4):351–381, 2017.
- [145] T. Chen, R. Bahsoon, and X. Yao. A Survey and Taxonomy of Self-Aware and Self-Adaptive Cloud Autoscaling Systems. *ACM Computing Surveys*, 51(3):61:1–61:40, 2018.
- [146] Prometheus Authors. Prometheus: Monitoring system and time series database. Cloud Native Computing Foundation.
URL: <https://prometheus.io/>
- [147] Kubernetes SIG Instrumentation. Kube-state-metrics. Kubernetes.
URL: <https://github.com/kubernetes/kube-state-metrics>
- [148] J. Schad, J. Dittrich, and J.A. Quiané-Ruiz. Runtime measurements in the cloud: observing, analyzing, and reducing variance. *Proceedings of the VLDB Endowment*, 3(1–2):460–471, 2010.
- [149] W. Iqbal, M. N. Dailey, D. Carrera, and P. Janecek. Adaptive resource provisioning for read intensive multi-tier applications in the cloud. *Future Generation Computer Systems*, 27(6):871–879, 2011.
- [150] R. Bouabdallah, S. Lajmi, and K. Ghedira. Use of Reactive and Proactive Elasticity to Adjust Resources Provisioning in the Cloud Provider. In *Proc. of the 18th IEEE International Conference on High Performance Computing and Communications*, pages 1155–1162, 2016.
- [151] A. Ali-Eldin, J. Tordsson, and E. Elmroth. An adaptive hybrid elasticity controller for cloud infrastructures. In *Proc. of the IEEE Symposium on Network Operations and Management*, pages 204–212, 2012.
- [152] B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal, and T. Wood. Agile dynamic provisioning of multi-tier Internet applications. *ACM Transactions on Autonomous and Adaptive Systems*, 3(1):1:1–1:39, 2008.
- [153] J. Jiang, J. Lu, G. Zhang, and G. Long. Optimal Cloud Resource Auto-Scaling for Web Applications. In *Proc. of the 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, pages 58–65. IEEE, 2013.

- [154] B. B. J.v. and D. Dharma. HAS: Hybrid auto-scaler for resource scaling in cloud environment. *Journal of Parallel and Distributed Computing*, 120:1–15, 2018.
- [155] P. Singh, A. Kaur, P. Gupta, S. S. Gill, and K. Jyoti. RHAS: robust hybrid auto-scaling for web applications in cloud computing. *Cluster Computing*, 24(2):717–737, 2021.
- [156] V. Rampérez, J. Soriano, D. Lizcano, and J. A. Lara. FLAS: A combination of proactive and reactive auto-scaling architecture for distributed services. *Future Generation Computer Systems*, 118:56–72, 2021.
- [157] N. S. Joshi, R. Raghuwanshi, Y. M. Agarwal, B. Annappa, and D. N. Sachin. ARIMA-PID: container auto scaling based on predictive analysis and control theory. *Multimedia Tools and Applications*, 83(9):26369–26386, 2024.
- [158] D. Zou, W. Lu, Z. Zhu, X. Lu, J. Zhou, et al. OptScaler: A Collaborative Framework for Robust Autoscaling in the Cloud. *Proceedings of the VLDB Endowment*, 17(12):4090–4103, 2024.
- [159] H. Fernandez, G. Pierre, and T. Kielmann. Autoscaling Web Applications in Heterogeneous Cloud Infrastructures. In *Proc. of the IEEE International Conference on Cloud Engineering*, pages 195–204, 2014.
- [160] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Prentice-Hall, Englewood Cliffs, New Jersey, 1984.
- [161] R. F. da Silva, R. Mayani, Y. Shi, A. R. Kemanian, M. Rynge, et al. Empowering Agroecosystem Modeling with HTC Scientific Workflows: The Cycles Model Use Case. In *Proc. of the 2019 IEEE International Conference on Big Data*, pages 4545–4552, 2019.
- [162] G. Juve, A. Chervenak, E. Deelman, S. Bharathi, G. Mehta, et al. Characterizing and profiling scientific workflows. *Future Generation Computer Systems*, 29(3):682–692, 2013.
- [163] A. J. Nebro, J. J. Durillo, and M. Vergne. Redesigning the jMetal multi-objective optimization framework. In *Proc. of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pages 1093–1100, 2015.

- [164] I. M. Ali, K. M. Sallam, N. Moustafa, R. Chakraborty, M. Ryan, et al. An Automated Task Scheduling Model Using Non-Dominated Sorting Genetic Algorithm II for Fog-Cloud Systems. *IEEE Transactions on Cloud Computing*, 10(4):2294–2308, 2022.
- [165] P. L’Ecuyer and E. Buist. Simulation in Java with SSJ. In *Proc. of the 37th Winter Simulation Conference*, pages 611–620, 2005.
- [166] C. Juiz, B. Capo, B. Bermejo, A. Fernández-Montes, and D. Fernández-Cerero. A Case Study of Transactional Workload Running in Virtual Machines: The Performance Evaluation of a Flight Seats Availability Service. *IEEE Access*, 11:81600–81612, 2023.
- [167] A. Van Renen, D. Horn, P. Pfeil, K. Vaidya, W. Dong, et al. Why TPC is Not Enough: An Analysis of the Amazon Redshift Fleet. *Proceedings of the VLDB Endowment*, 17(11):3694–3706, 2024.
- [168] 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*, 41(1):23–50, 2011.