Location-Aware Application Deployment in Multi-Cloud

by

Tao Shi

A thesis
submitted to the Victoria University of Wellington
in fulfilment of the
requirements for the degree of
Doctor of Philosophy
in Computer Science.

Victoria University of Wellington
2022

Abstract

The client-centric multi-cloud has become a popular cloud ecosystem because it allows enterprise users to share the workload across multiple cloud providers to achieve high-quality services with lower operation costs and higher application resilience. From the perspective of application providers, the location of cloud resources for application deployment significantly impacts the deployment cost and performance of applications, e.g., request response time. This gives rise to the problem of location-aware application deployment in multi-cloud to select suitable cloud resources from widely distributed multi-cloud data centers to balance the cost and performance. Existing research works did not pay full attention to the key impact of the location for application deployment. Therefore, it is urgent to study the problem both theoretically and in practice. In this thesis, innovative optimization methods and machine learning techniques are proposed for three common scenarios, namely composite application deployment, application replication and deployment, and elastic application deployment.

First, this thesis studies the composite application deployment problem with the goal to minimize the average response time of composite applications subject to a budget constraint. We propose a Hybrid Genetic Algorithm (GA)-based approach, i.e., *H-GA*, for solving the problem with an extremely large search space. *H-GA* features a newly designed and domain-tailored service clustering algorithm, repair algorithm, solution representation, population initialization, and genetic operators. Experiments show that *H-GA* can outperform significantly several state-of-the-art approaches, achieving up to about 8% performance improvement in terms of response time, and 100% budget satisfaction in the meantime.

Second, this thesis studies the application replication and deployment problem with the goal to minimize the total deployment cost of all application replicas subject to a stringent requirement on average response time. We propose two approaches under different optimization frameworks to solve the problem. With user requests dispatched to the closest application replicas, we develop an approach under a GA framework for Application Replication and Deployment (ARD), i.e., *GA-ARD*. *GA-ARD* features problem-specific solution representation, fitness measurement, and population initialization, which are effective to optimize the deployment of application replicas in multi-cloud. The experiments show that *GA-ARD* outperforms common application replication and placement strategies in the industry. With user requests flexibly dispatched among different application replicas, we develop another approach under a two-stage optimization framework, i.e., *MCApp*. *MCApp* can optimize both replica deployment and request dispatching by combining mixed-integer linear programming with domain-tailored large neighborhood search. Our experiments show that *MCApp* can achieve up to 25% reduction in total deployment cost compared with several recently developed approaches.

Third, this thesis studies the elastic application deployment problem to minimize the deployment cost over a time span such as a billing day while satisfying the constraint on average response time. The goal of adapting resources for application deployment in response to dynamic and distributed workloads motivates us to adopt deep reinforcement learning (DRL) techniques. The proposed approach, namely *DeepScale*, applies a deep Q-network (DQN) to capture the optimal scaling policy that can perform online resource scaling. *DeepScale* also includes a long short term memory-based prediction model to allow the DQN to consider predicted future requests while making cost-effective scaling decisions. Besides, we design a penalty-based reward function and a safety-aware action executor to ensure that any scaling decisions made by DRL can satisfy the response time constraint. The experiments show that *DeepScale* can sig-

nificantly reduce the deployment cost of applications compared with the state-of-the-art baselines, including Amazon auto-scaling service and recently proposed RL-based algorithms. In the meanwhile, *DeepScale* can effectively satisfy the constraint on the average response time.

In summary, this thesis studies three new problems for location-aware application deployment in multi-cloud. We propose four novel approaches under different optimization and machine learning frameworks, i.e., *H-GA*, *GA-ARD*, *MCApp*, and *DeepScale*, for solving these problems. New constraint handling techniques are developed to satisfy the practical deployment requirements of enterprise applications.

Acknowledgments

I would like to express my sincere gratitude to those who gave me the assistance and support during my Ph.D. study.

My deepest gratitude goes to my supervisors, A/Prof Hui Ma and Dr. Gang Chen. A/Prof Hui Ma has spent dedicated time and efforts to train my research skills, and provided constructive and challenging feedback to improve my research work. This thesis would not have been possible without her encouragement, motivation, inspiration, and guidance. Dr. Gang Chen is always nice to talk to, and the discussions with him always bring me to further thinking about my research. I am also grateful to him for helping me improve my research writing skills.

I am grateful for the Victoria Doctoral Scholarship, the Marsden Fund of New Zealand (VUW1510), the University Research Fund, School of Engineering and Computer Science and Faculty of Engineering at Victoria University of Wellington for their financial support over the past three years.

Thank Prof Sven Hartmann gave his feedback on my articles fast and helpful. Also, I wish to thank my friends in the Evolutionary Computation Research Group (ECRG) for creating an active and helpful research environment. Thank you to Boxiong Tan and Victoria Huang for discussing and sharing ideas.

Last but not least, I wish to thank my family for their love, encouragement, and support.

List of Publications

- **Tao Shi**, Hui Ma, Gang Chen, and Sven Hartmann, "Cost-Effective Web Application Replication and Deployment in Multi-Cloud Environment," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 8, pp. 1982–1995, 2022.

- **Tao Shi**, Hui Ma, Gang Chen, and Sven Hartmann, "Location-Aware Service Brokering in Multi-Cloud via Deep Reinforcement Learning", *2021 International Conference on Service-Oriented Computing (ICSOC)*, Springer, Cham, 2021, 756-764.

- **Tao Shi**, Hui Ma, Gang Chen, and Sven Hartmann, "Location-Aware and Budget-Constrained Service Deployment for Composite Applications in Multi-Cloud Environment," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 8, pp. 1954-1969, 2020.

- **Tao Shi**, Hui Ma, Gang Chen, and Sven Hartmann, "Location-Aware and Budget-Constrained Application Replication and Deployment in Multi-Cloud Environment," *2020 IEEE International Conference on Web Services (ICWS)*, IEEE, 2020, 110-117.

- **Tao Shi**, Hui Ma, and Gang Chen, "Seeding-Based Multi-Objective Evolutionary Algorithms for Multi-Cloud Composite Applications Deployment," *2020 IEEE International Conference on Services Computing (SCC)*, IEEE, 2020, 240-247.

- **Tao Shi**, Hui Ma, and Gang Chen, "Divide and conquer: Seeding strategies for multi-objective multi-cloud composite applications deployment," *2020 Genetic and Evolutionary Computation Conference Companion (GECCO)*, ACM, 2020, 317-318.

- **Tao Shi**, Hui Ma, and Gang Chen, "A Genetic-Based Approach to Location-Aware Cloud Service Brokering in Multi-Cloud Environment," *2019 IEEE International Conference on Services Computing (SCC)*, IEEE, 2019, 146-153.

- **Tao Shi**, Hui Ma, and Gang Chen, "A Seeding-based GA for Location-Aware Workflow Deployment in Multi-cloud Environment," *2019 IEEE Congress on Evolutionary Computation (CEC)*, IEEE, 2019, 3364-3371.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem Statement

Gartner[1] forecasts that worldwide end-user spending on public cloud services will approach 400 billion U.S. dollars in 2022 and maintain rapid growth. According to Forbes [37], currently, 83% of enterprise workloads have been moved to the cloud, and half of them will be hosted on public cloud platforms.

The thriving public cloud market drastically affects its participants. On the one hand, the competitive *cloud providers* are progressively increasing investment and launching various services with customized properties, pricing models, and Service Level Agreements (SLAs) in different geographic locations[2]. On the other hand, *cloud users* tend to apply cloud services from multiple cloud providers to achieve two major benefits as follows.

- *Cost saving:* By utilizing multiple clouds, cloud users can dynamically distribute workload among cloud providers freely to avoid the changes in policy and pricing [59]. In the context of the competitive

[1]https://www.gartner.com/

[2]https://azure.microsoft.com/en-us/global-infrastructure/regions/

cloud marketplace, cloud users can achieve substantial cost savings.

- *Application resilience:* Due to the potential unavailability of services from one single cloud provider, the Berkeley view of cloud computing advises the use of multiple clouds in order to achieve high resilience [46], [210].

There are other potential benefits for the usage of multiple clouds, such as access to widely distributed and legislation-compliant services [22] and avoidance of vendor lock-in [53], [59].

In cloud computing, cloud users can be either the *end-users* who use the cloud resources directly or the *application providers* who have their own users [45]. Recently, application providers are interested in embracing *multi-cloud*, because the *client-centric* paradigm does not require cloud providers to adopt and implement standard interfaces, protocols, formats, and architectural components [178]. That is, multi-cloud allows application providers to manage resources across multiple clouds [131]. Usually, an *adapter* layer with different Application Programming Interfaces (APIs) is required to access resources from multiple clouds as shown in Figure 1.1.

In this thesis, we focus on the perspective of *application providers* and clarify the following terms for reference.

- *Application providers:* entities that offer domain-specific software or applications to end users by using multi-cloud resources made available by cloud providers.

- *Application services (or applications for short):* specific functionalities of software or applications, e.g., the meteorological service of New Zealand, MetService[3], which we refer to as cloud-based services.

The network latency between end-users and cloud-based services in different locations significantly affects the performance of applications [68],

[3]https://www.metservice.com/

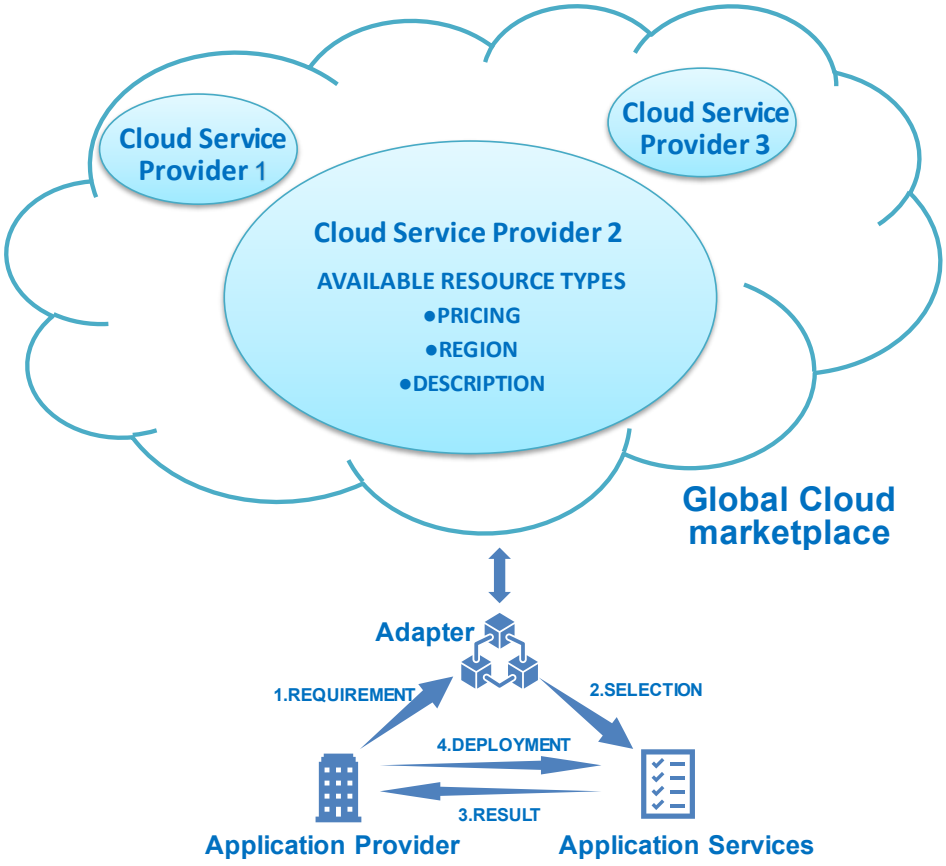


Figure 1.1: A simplified application deployment system in multi-cloud based on [178].

especially for interactive applications [59]. Besides, the prices of cloud resources in different regions can vary substantially. For example, the prices of m6g.large (Linux) from Amazon EC2[4] are $0.077 and $0.1224 per hour in Northern Virginia (USA) and Sao Paulo (Brazil) respectively. That is, the *location* of cloud resources for application deployment significantly impacts both the *deployment cost* and *performance* of applications [69], [154], [155], [156], [157], [158], [159]. This gives rise to the problem of *location-*

[4]https://aws.amazon.com/ec2/pricing/on-demand/

aware application deployment in multi-cloud. This problem aims to select suitable cloud resources from widely distributed multi-cloud data centers to balance the cost and performance. Existing research works did not pay full attention to the key impact of the location for application deployment [135]. Therefore, it is urgent to study the problem both theoretically and in practice. Particularly, three important scenarios for the *location-aware* application deployment in multi-cloud need to be investigated.

- *Composite Application Deployment (CAD)*. There is an ongoing trend for application providers to offer several business applications simultaneously, and the relevant business processes are usually abstracted as workflows [218]. Based on Service-Oriented Architecture (SOA) [28], these composite applications consist of a set of specific functionalities, i.e., constituent services. Because different composite applications have different user distributions, constituent services can be deployed in different *locations* to reduce network latency. Besides, application providers are often interested in budgetary control to ensure that their actual costs adhere closely to their financial plan [25]. The budgetary control emphasizes the optimization of application performance within a given budget, which is of immense practical significance for application providers [142]. However, the existing works on *CAD* usually deploy constituent services in the same location and do not consider the budget impact.

- *Application Replication and Deployment (ARD)*. For some applications, a low average response time must be satisfied to guarantee the quality of experience (QoE) [104], [201]. *ARD* applies the scenario where application providers must replicate applications in multiple *locations* to ensure the stringent requirement on average response time. In multi-cloud, *ARD* needs to resolve two key issues: (1) How should the types and locations of resources be selected to deploy application replicas so that the total deployment cost is minimized? (2) How

should user requests be distributed among application replicas such that the average response time can meet the requirement?

- *Elastic Application Deployment (EAD)*. Enterprise applications often process dynamic workloads from the worldwide user community [6]. These applications should dynamically acquire and release resources to handle the varying workloads. The elasticity can improve the cost-effectiveness of application deployment while satisfying the constraint on average response time [201]. In the literature, *EAD* was usually realized by scaling resources within a single data center [141], [213]. In this thesis, *EAD* aims to efficiently adapt the multi-cloud resources in different *locations* for application deployment in response to highly dynamic and widely distributed workloads.

1.2 Motivations

Upon considering a large number of available cloud resources provided by multiple cloud providers at different locations with different prices, solving *CAD* becomes a very challenging task. Genetic Algorithm (GA) is a classic optimization technique to generate high-quality solutions for various combinatorial optimization problems with practical importance [39], [197]. In particular, driven by a population-based solution improvement framework, GA has been widely used to select proper multi-cloud resources for scalable cloud application deployment [69], [85], [211]. However, new technical innovations are required to effectively apply GA to address *CAD* in multi-cloud due to three reasons. (1) We should consider the geographical location of cloud resources. In the global cloud marketplace, a large number of different resources are provided at different data centers. The large and complex search space can impair the scalability and applicability of the existing GA-based approaches [61]. (2) Many of GA-based algorithms [99], [208] solve constrained optimization problems by intro-

ducing penalty functions. However, numerous research works in the literature have shown that repair algorithms have the potential to outperform penalty-based algorithms in GA by adopting problem-specific heuristics [36], [146]. We need to propose a specific repair algorithm to solve our problem effectively because there are no standard guidelines for the design of repair algorithms [111]. (3) To control the computation time of the repair process, we should also consider the efficiency of our proposed repair algorithm [146].

Considering application replicas, *ARD* in multi-cloud is similar to the cloud-hosted data placement and replication problems [87], [114]. Integer Linear Programming (ILP) has been considered as the dominant method to model these problems [66], [201]. However, *ARD* considers the response time of user requests, including both the network latency and the request processing time. For the applications under high workloads, the request processing time is nonlinearly related to the capacity of deployed cloud resources and users' demands [47]. The nonlinear nature renders the ILP-based approaches in [66], [201] inapplicable. Many meta-heuristic algorithms have also been used for the data placement and replication problem in clouds [88], [102]. However, these approaches focused on the location selection for data replicas, while *ARD* must select both the locations and the types of cloud resources.

EAD is a dynamic scenario to scale multi-cloud resources for application deployment. Finding a cost-effective and performance-satisfactory deployment solution in multi-cloud involves a complex search space. The existing algorithms for *EAD* in multi-cloud (e.g., using meta-heuristics in [6]) can incur high computational costs to obtain adaptive solutions. Moreover, these algorithms usually do not consider the impact of current deployment decisions on the future deployment cost and response time. The goal of adapting resources for application deployment in response to dynamic and distributed workloads motivates us to adopt Deep Reinforcement Learning (DRL) techniques [173]. DRL applies a deep neural

network, e.g., Deep Q-Network (DQN), to model the optimal policy for scaling cloud resources. The adaptive nature of DRL makes it very appealing to devise effective scaling policies for applications with dynamic and widely distributed workloads. However, the existing RL-based algorithms, e.g., [141], [213], cannot be directly applied to solve the *EAD* problem because they lack the safety mechanisms to satisfy the response time constraint.

1.3 Research Goals

The overall goal of this research is to achieve *location-aware application deployment in multi-cloud* effectively and efficiently. More specifically, this research deals with the three application deployment scenarios, i.e., *CAD*, *ARD*, and *EAD*.

1. *CAD* deals with the deployment of a set of composite applications in multi-cloud to minimize the average response time subject to a budget constraint. This is the scenario where application providers have the demand for budget management during composite application deployment. Considering the deployment location of constituent services and budget constraints, we will formulate the new *CAD* problem and propose a hybrid GA-based approach to solve the problem.

   (1) *Formulate a more widely applicable problem for CAD in multi-cloud.* To achieve a high level of control and flexible management goals [81], we consider the *Infrastructure-as-a-Service (IaaS)* paradigm, i.e., deploying composite applications on a set of *Virtual Machines (VMs)*. In this formulation, we will formally define (a) the composite applications consisting of constituent services, (b) the multi-cloud application deployment system including the *capacity-feasible* VMs (i.e., the VM types whose capacities are

greater than their workloads), (c) the *budget* constraint and the objective. Because the deployed applications need to process tens of thousands or even millions of requests daily [205], we model the processing of ongoing application requests through a *queuing model* to precisely capture the performance of applications.

(2) *Propose a hybrid GA-based approach for the CAD problem.*

First, the research aims to investigate a *service clustering* mechanism based on the dependency among services to reduce the complexity/size of the search space, thereby ensuring the efficiency and effectiveness of the GA-based approach.

Second, the research aims to design a problem-specific *repair algorithm* to solve the constrained optimization problem. The repair algorithm will progressively downgrade the service hosting VM types to meet the budget requirements at the expense of low-performance deterioration.

Finally, the research aims to achieve a desirable trade-off between performance and computation time during the repair process. We will design an *adaptive bound* to choose and transform only a portion of over-budget deployment solutions into constraint-compliant solutions.

2. *ARD* aims to minimize the total deployment cost of applications subject to the stringent performance requirements in terms of average response time. Considering the deployment location of application replicas, the deployment cost and average response time of applications depend on both the replica deployment plan (i.e., how to deploy all application replicas on specific resources in specific multi-cloud data centers) and the request dispatch plan (i.e., how to dispatch user requests among all application replicas). We will formulate the new *ARD* problem with the two matching plans and pro-

pose two approaches under different optimization frameworks for the problem.

(1) *Formulate a more widely applicable problem for ARD in multi-cloud.*

ARD focuses on the *IaaS* paradigm, i.e., deploying application replicas on a large group of VMs. In this formulation, we will formally define (a) two location-related plans, i.e., the replica deployment plan and request dispatch plan, (b) the *data consistency model* for application replicas, (c) the *performance* constraint in terms of the average response time and the cost objective. Depending on user requests dispatched to the closest application replicas (a common practice) or flexibly dispatched among different application replicas (more cost-effective), *ARD* is further defined as *ARD with close dispatching* problem and *ARD with flexible dispatching* problem, correspondingly.

(2) *Propose a two-level optimization approach under the GA framework for the ARD with close dispatching problem.*

Dispatching user requests to the closest application replicas or services is a common practice in the literature [74], [171] and industry [113]. With close request dispatching, the location selection of application replicas has a major impact on the performance and deployment cost of applications.

First, the research aims to optimize *location* selection for application replicas in multi-cloud by GA (first level optimization).

Second, the research aims to design a problem-specific heuristic to perform *VM types* selection based on the locations of application replicas (second level optimization). Particularly, after deploying all application replicas to the cheapest capacity-feasible VMs, we will progressively upgrade the VM types to reduce the response time as long as the performance requirement is satisfied.

Finally, *seeding strategies* have been proven to be an effective way to improve the GA-based algorithms by injecting knowledge about the problem [27]. Therefore, the research aims to investigate a seeding strategy for the GA-based approach to improving solution quality and convergence speed.

(3) *Propose a two-stage optimization approach for the ARD with flexible dispatching problem.*

The flexible request dispatching enables further reduction of the deployment cost by simultaneously optimizing the replica deployment and request dispatching [201]. With the extra optimization decision, i.e., the request dispatch plan, we propose a two-stage optimization approach combining Mixed-Integer Linear Programming (MILP) with domain-tailored Large Neighborhood Search (LNS).

First, the research aims to *linearize* the *ARD with flexible dispatching* problem so that it can be solved by MILP methods. Because adapting upper bounds on VMs' utilization rate and average response time helps to reduce the total deployment cost, the research aims to propose a MILP-based algorithm to obtain efficiently a high-quality *base* solution. The performance of the base solution will be improved by adaptively updating the upper bounds on both VMs' utilization rate and average response time.

Second, the research aims to develop an LNS-based algorithm to improve the base solution. To build an effective LNS process, a new *destroy heuristic* and a new *repair heuristic* will be designed to adjust the replica deployment plan. Based on the new replica deployment plan, we will propose a *delay-oriented heuristic* to efficiently dispatch user requests for the purpose of low average response time.

3. *EAD* aims to automatically adapt multi-cloud resources in response to the varying and distributed application workloads. For this scenario, the total cost for the application deployment over a time span such as a billing day needs to be minimized while satisfying the constraint on average response time. We will formulate the new *EAD* problem and propose a novel approach to solve the problem.

   (1) *Formulate a more widely applicable problem for EAD in multi-cloud.*

   Currently, elastic deployment of applications increasingly relies on *containers*, an industry-leading lightweight virtualization technology [18], [140]. By bundling together an application with all its dependencies (e.g., libraries and code), containers lay the technical foundation for fast and easily adapting the application deployment through *horizontal and vertical scaling*. The horizontal scaling in multi-cloud can be realized by increasing and decreasing the number of application replicas, i.e., containers, at different locations. The vertical scaling aims to increase and decrease the number of resources assigned to any application replica. In this formulation, we will formally define (a) the *dynamic* scenario in terms of application workloads, (b) two types of container-based scaling in multi-cloud, (c) the objective regarding the total deployment cost over a time span and the constraint on average response time.

   (2) *Propose a novel DRL-based approach with a Long Short Term Memory (LSTM)-based prediction model for the EAD problem.*

   First, because the start-up and shut-down time of containers is short, container scaling requires higher timeliness and accuracy [213]. The research aims to train an *LSTM-based workload predictor* based on request arrival history and a DRL-based scaling policy considering the predicted future workloads. Based on the predicted workload, the trained scaling policy will perform

more effective container scaling in multi-cloud with a lower total deployment cost.

Second, the research aims to propose a *penalty*-based reward function to train scaling policies toward the constraint-compliant scaling and design a *safety-aware* action executor to ensure that any scaling decisions made by the scaling policy will satisfy the performance constraint.

1.4 Major Contributions

This thesis proposes five major contributions to the area of cloud-based application deployment:

1. Due to the key impact of the deployment location, this thesis defines three new problem formulations for location-aware application deployment in multi-cloud. To the best of our knowledge, this is the first study in the literature on cloud-based application deployment considering the location impact on both the cost and performance of applications on the global scale. That is, the existing problem formulations on the three scenarios did not consider the locations of cloud resources as optimization decisions for application deployment or only considered the cloud resources within one or two data centers. For the three different scenarios, i.e., *CAD*, *ARD*, and *EAD*, different deployment paradigms, deployment objects, constraints, and objectives are included in the new problem formulations.

2. This thesis proposes a hybrid GA-based approach, namely *H-GA*, to the *CAD* problem. H-GA has a newly designed and domain-tailored service clustering method to reduce the size of the search space due to considering the locations of multi-cloud resources and a repair algorithm to guarantee the total deployment cost within the given

budget. To evaluate the proposed approach, we collect the information about VM capacity and pricing offered by the global top three cloud providers, i.e., Amazon, Microsoft, and Alibaba. Based on the collected information, extensive empirical study has been conducted to deploy a large group of composite applications with different service diversities and budget factors. The experimental results indicate that *H-GA* significantly outperforms several state-of-the-art algorithms. Part of this contribution has been published in:

Tao Shi, Hui Ma, Gang Chen, and Sven Hartmann, "Location-Aware and Budget-Constrained Service Deployment for Composite Applications in Multi-Cloud Environment," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 8, pp. 1954-1969, 2020.

3. This thesis proposes a novel GA-based approach to the *ARD with close dispatching* problem, namely *GA-ARD*. *GA-ARD* has a newly designed solution representation, fitness measurement, and population initialization. Experimental results show *GA-ARD* can significantly reduce deployment cost compared with the industry-leading application replication and placement strategies. Part of this contribution has been published in:

   Tao Shi, Hui Ma, Gang Chen, and Sven Hartmann, "Location-Aware and Budget-Constrained Application Replication and Deployment in Multi-Cloud Environment," *2020 IEEE International Conference on Web Services (ICWS)*, IEEE, 2020, 110-117.

4. This thesis proposes a novel approach, named *MCApp*, to solve the *ARD with flexible dispatching* problem. *MCApp* creates a hybrid optimization process that combines an iterative MILP-based algorithm and a domain-tailored LNS-based algorithm to simultaneously optimize the replica deployment plan and the request dispatch plan. *MCApp* is compared to several state-of-the-art approaches and has

achieved superior performance on many problem instances with varied applications. Part of this contribution has been published in:

Tao Shi, Hui Ma, Gang Chen, and Sven Hartmann, "Cost-Effective Web Application Replication and Deployment in Multi-Cloud Environment," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 8, pp. 1982–1995, 2022.

5. This thesis proposes a novel DRL-based approach, i.e., *DeepScale*, for the *EAD* problem. *DeepScale* features an LSTM-based prediction model, a newly designed safety-aware action executor, and penalty-based reward function. We develop a fully functioning prototype of *DeepScale* using PyTorch [130] that can be directly applied to real-world multi-cloud. Extensive empirical study of *DeepScale* has been performed. The experimental results show that *DeepScale* outperforms Amazon auto-scaling service[5] and the recently proposed baselines.

1.5 Organization of Thesis

- *Chapter 1: Introduction*

  This chapter includes the problem statement, motivations, research goals, contributions, and thesis organization.

- *Chapter 2: Background and Literature Review*

  This chapter presents a background of multi-cloud, distributed application in cloud, optimization, and machine learning techniques we intend to use. In addition, it presents a literature review of *CAD*, *ARD*, and *EAD* with highlighted limitations and challenges.

- *Chapter 3: Composite Application Deployment*

[5]https://aws.amazon.com/cn/autoscaling/

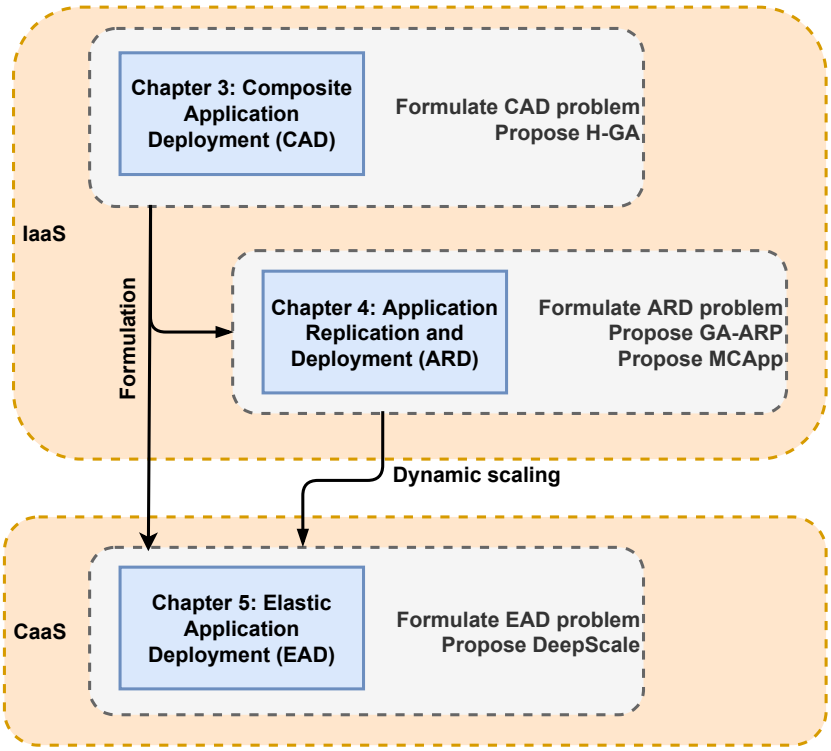


Figure 1.2: The connection between major contributions chapters in the thesis (see Subsection 2.1.1 for details about IaaS and CaaS).

This chapter formulates a novel *CAD* problem. A hybrid GA-based approach, i.e., *H-GA*, is proposed to solve the problem. The performance of *H-GA* is compared with the baseline algorithms using real-world datasets.

- *Chapter 4: Application Replication and Deployment*

  This chapter formulates a novel *ARD* problem. Then, two novel approaches, i.e., *GA-ARD* and *MCApp*, are respectively proposed to handle the *ARD with close dispatching* problem and the *ARD with flexible dispatching* problem. The performance of the two approaches is

compared with the baseline algorithms using real-world datasets.

- *Chapter 5: Elastic Application Deployment*

  This chapter formulates a novel *EAD* problem. It proposes a novel DRL-based approach with an LSTM-based prediction model, i.e., *DeepScale*, to solve this problem. The performance is evaluated and compared with baseline algorithms using real-world datasets.

- *Chapter 6: Conclusions and future works*

  In this chapter, the conclusions and findings in each chapter are presented and summarized. The chapter also describes the main future research directions arising from the contributions of this work.

The connection between the major contribution chapters in this thesis is shown in Figure 1.2. The *CAD* problem in Chapter 3 provides the foundation for formulating the multi-cloud application deployment. This formulation is extended to support the *ARD* problem and *EAD* problem in Chapters 4 and 5. Meanwhile, *H-GA* is proposed for the *CAD* problem. Considering application replication, Chapter 4 develops two approaches, i.e., *GA-ARD* and *MCApp*, for the *ARD* problem. Chapter 5 proposes *DeepScale* to solve the *EAD* problem considering dynamic scaling.

Chapter 2

Background and Literature Review

This chapter introduces the fundamental concepts of application deployment in multi-cloud, optimization and machine learning techniques, and related work. Section 2.1 introduces the concepts of cloud computing, interconnected cloud paradigms, and distributed application in cloud. Section 2.2 explains the optimization and machine learning techniques that we intend to use for the problem of location-aware application deployment in multi-cloud. Section 2.3 reviews the related work about Composite Application Deployment (*CAD*), Application Replication and Deployment (*ARD*), and Elastic Application Deployment (*EAD*). Section 2.4 concludes the findings in the literature and positions our research in the field.

2.1 Application Deployment in Multi-cloud

In this section, we first discuss the stakeholders in cloud computing from a resource management perspective and the types of services offered in cloud. Then, Subsection 2.1.2 discusses the different paradigms to realize interoperability among multiple clouds. Subsection 2.1.3 introduces

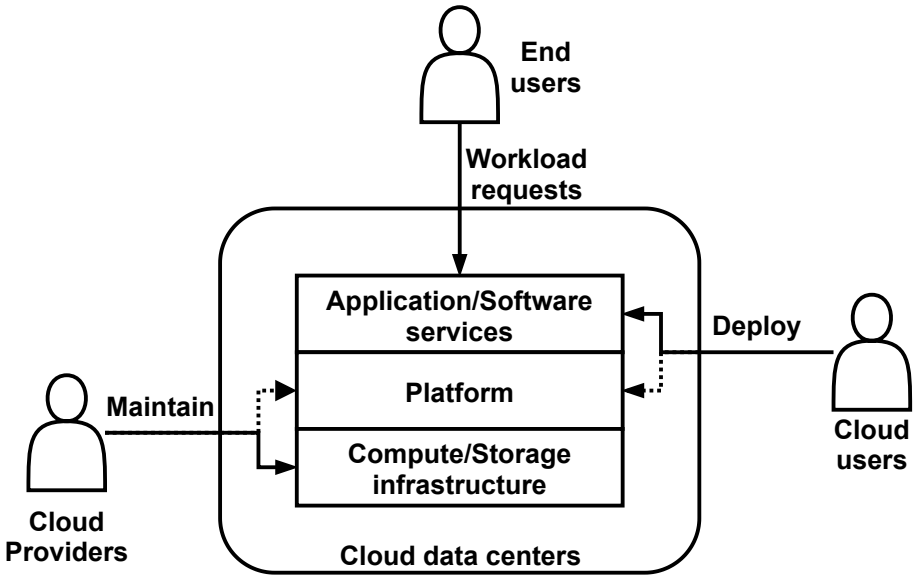


Figure 2.1: Key stakeholders in cloud computing adapted from [83].

some commonly deployed distributed applications in cloud. Subsection 2.1.4 categorizes the problem of location-aware application deployment in multi-cloud into three scenarios, i.e., *CAD*, *ARD*, and *EAD* because each of them must be solved by effective and efficient methods.

2.1.1 An Overview of Cloud Computing

Cloud computing is the delivery of different computing services through the Internet with pay-as-you-go pricing [112]. According to NIST's definition [125], *"cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction."* Cloud computing has become increasingly popular for a number of reasons including cost savings, increased productivity, speed and efficiency, performance, and security [112].

From a resource management perspective, cloud computing involves three key stakeholders, i.e., cloud providers, cloud users, and end-users (see Figure 2.1). *Cloud providers* build data centers and maintain the cloud servers at the data centers. *Cloud users*, e.g., application providers, deploy their applications or software to these cloud servers from anywhere in the world. *End users* can use the applications without installing them on their local computers once the applications are deployed. Cloud providers charge fees from cloud users for using the infrastructure, i.e., application deployment cost.

Cloud computing is primarily comprised of three types of services: *Software-as-a-Service (SaaS), Platform-as-a-Service (PaaS), and Infrastructure-as-a-Service (IaaS)*.

- SaaS delivers software applications over the Internet, on-demand, and typically on a subscription basis. With SaaS, cloud providers host and manage the software applications and underlying infrastructure, and handle any maintenance, like software upgrades and security patching [112]. The cloud providers are also the application providers (i.e., cloud users in Figure 2.1) [83]. This type of service can be found in Microsoft Office 365 [186].

- PaaS supplies an on-demand platform for developing, testing, delivering, and managing software applications. PaaS is designed to set up or manage the underlying infrastructure of servers, storage, network, and databases needed for application development. Google AppEngine is an example of PaaS [186].

- IaaS delivers cloud computing infrastructure, i.e., servers and Virtual Machines (VMs), storage, networks, operating systems, on a pay-as-you-go basis. After that, cloud users manage the VM instances as remote servers, while cloud providers do very little management inside the VM instances. Amazon Elastic Compute Cloud (EC2) is one example of IaaS [186].

In recent years, *Containers-as-a-service (CaaS)* has become a popular cloud service type [78]. CaaS can manage and deploy applications using container-based abstraction. Containers are lightweight packages of application code together with dependencies such as specific versions of programming languages and libraries required to run application services. CaaS is often termed as a subset of IaaS with different functionalities [78]. Google Kubernetes and Docker Swarm are two examples of CaaS.

In practice, application providers often prefer IaaS in order to achieve a high level of control and flexible management goals [42], [174]. Therefore, we consider the IaaS for the *CAD* and *ARD* scenarios in this thesis. Due to the lightweight and flexible characteristics, elastic development and run-time management of applications increasingly rely on containers [18], [140]. Therefore, we consider the CaaS for the *EAD* scenario.

2.1.2 Interconnected Cloud Paradigms

According to the survey from Gartner [57], 81% of public cloud users are using multiple clouds. Interoperability across multiple clouds is important for the realization of computing as a utility like other utilities such as electricity and telephony. In practice, cloud interoperability can be obtained through either a *provider-centric* approach or a *client-centric* approach [178].

The provider-centric approach requires cloud providers to adopt and implement standard interfaces, protocols, formats, and architectural components [131]. The scenarios based on the provider-centric approach include hybrid cloud and federated cloud paradigms [178].

- A *hybrid cloud* allows a private cloud to form a partnership with a public cloud. By the hybrid cloud, an application run in a private data center can burst into a public cloud when the demand for computing capacity spikes.

- A *federated cloud* assumes a prior business agreement among cloud providers and they collaborate to exchange resources voluntarily [59]. In this paradigm, cloud providers aim to overcome resource limitations in their local infrastructure by outsourcing workloads to other members of the federation. Besides, the federated cloud allows cloud providers with low resource utilization to lease part of their resources to other federation members to avoid wasting the nonstorable compute resources.

In practice, one comprehensive set of standards for the provider-centric approach is difficult to develop and adopt by all providers. However, even if the *provider-centric* approach is not supported by cloud providers, cloud users are still able to benefit from the *client-centric* interoperability facilitated by third-party brokers or user-side libraries corresponding to the following two paradigms [178].

- *Aggregated service by broker* offers an integrated service to cloud users by coordinating access and utilization of multiple cloud resources [137].

- In the *multi-cloud* paradigm, application providers are responsible to manage resources across multiple clouds. The application providers may require an adapter layer with different APIs to run services on different clouds [178]. In multi-cloud, application providers perform application deployment, negotiate with each cloud provider, and monitor each cloud provider during the service operation.

In this thesis, we consider the multi-cloud paradigm because it gives application providers the freedom to use the best possible cloud for each workload [77], [172]. For example, to achieve cost-saving, application providers can dynamically distribute application workload among different cloud providers to deal with the changes of policies and pricing [178].

2.1.3 Distributed Application in Cloud

Distributed applications deployable in cloud can be divided into batch processing applications and interactive applications [59].

Batch processing applications allow end users to submit and execute jobs without further input. Therefore, they are also known as job-based applications [59]. Particularly, batch processing applications can be further classified as singular-job applications or periodical-job applications.

- *Singular-job applications* execute each job only once if no failure occurs. Most of these applications fall into the High Performance Computing (HPC), High Throughput Computing (HTC) or Many-Task Computing (MTC) categories [136]. Scientific workflows [16] also can be considered as singular-job applications composed of multiple inter-dependent sub-jobs [59].

- *Periodical-job applications* repeatedly execute jobs over a period of time. Periodic Extract Transform Load (ETL) jobs in data warehouses, big data analytical jobs, and corn jobs [92] are the examples of periodical-jobs [59].

Interactive applications are also known as online applications, which constantly interact with end-users. Most interactive applications are data-centric or compute-intensive [59]. Because interactive applications must be constantly available, they will benefit the most from the cost-saving and high resilience of multi-cloud [59]. Besides, many interactive applications serve end-users from around the world. Therefore, they can benefit from the geographical diversity of multi-cloud data centers [59]. Because the interactive application deployment in multi-cloud has seldom been well studied in the literature, we study the problem in this thesis. In the following content, we use applications to refer to interactive applications.

2.2 Optimization and Machine Learning Techniques

This section reviews the optimization and machine learning techniques that are applied to our work. Because the *CAD*, *ARD*, and *EAD* problems are all constrained optimization problems, different constraint handling techniques are also reviewed in this section.

2.2.1 Evolutionary Computation

As a sub-domain of artificial intelligence, Evolutionary Computation (EC) covers the majority of algorithms inspired by biological evolution [14]. EC is a promising approach for effective resource management in cloud computing, such as scheduling [38], [48], [73], mapping [150], [151], load balancing and capacity planning [61]. Genetic Algorithm (GA) is one of the most popular EC techniques. It can solve the problems with vast search space and avoid being trapped into local optima by adopting a population-based solution improvement framework [115]. More importantly, GA can be used to solve problems that cannot be tackled by using traditional gradient-based techniques [197].

Genetic Algorithms (GAs)

GA is a meta-heuristic originally developed by Holland [71]. A typical GA requires a genetic representation of the solution with the help of some encoding methods. Each chromosome is associated with a fitness value based on a fitness function, and the fitness value of each individual is an indication of its chances of survival and reproduction in the next generation.

A general GA procedure begins with an *initialization* of a population of chromosomes. The initialization process generates a population of solutions that widely spread across the solution space. The purpose of ini-

Algorithm 1 Typical GA approach [39]

```
1: Initialize a population of chromosome randomly
2: Evaluate population with fitness function
3: while stopping criterion not met do
4:    Apply Selection
5:    Apply Crossover
6:    Apply Mutation
7:    for all offspring chromosome do
8:       Evaluate population with fitness function
9:    end for
10: end while
11: Return the best solution in the final generation;
```

tialization is to start the search in a good position by obtaining knowledge through sampling. During the process of evolution, three basic GA operations, i.e., *selection*, *crossover*, and *mutation*, will be performed repeatedly until any given stopping condition is met (see Algorithm 1).

In multi-cloud, there are a large number of different VM types available at different data centers. Therefore, the *CAD* problem and the *ARD with close dispatching* problem involve an extremely large search space. Compared with heuristic-based algorithms, e.g., BHEFT [216], GA-based algorithms are more competent for solving the two problems. Although the GA-based algorithms do not guarantee the global optimal solution, they usually can find near-optimal solutions within a feasible amount of time [43], [197].

Constraint Handling Techniques

GAs are unconstrained search techniques [34]. There are many studies regarding the mechanisms that allow GAs to deal with equality and inequality constraints.

The most common way of incorporating constraints into a GA is to

introduce penalty functions [146]. The idea of *penalty functions* is to transform a constrained optimization problem into an unconstrained one by adding (or subtracting) a certain value to/from the objective function based on the amount of constraint violation in a certain solution. The types of penalty functions used with GAs include death penalty, static penalty, dynamic penalty, adaptive penalty [52], [206].

Numerous research works in the literature have shown that repair algorithms have the potential to outperform penalty functions for GAs to handle constraints [36], [146]. The *repair algorithms* consist in devising a procedure (or mechanism) that allows transforming an infeasible solution into a feasible one, i.e., to repair the infeasible individual. Such repaired solutions can be only used for the fitness evaluation. The repaired solutions can also replace the original solutions in the population (with some probability). This approach is problem-dependent since a specific repair algorithm has to be designed for each particular problem [34], [52].

Other constraint-handling techniques used with GAs include special representations and separation of constraints and objectives [34]. Particularly, special representations apply to the scenario where traditional generic representation schemes might not be appropriate to tackle the target problems. Separation of constraints and objectives handles constraints and objectives separately by coevolution [129], assigning a higher fitness to feasible solutions [133], behavioural memory [147], and using multi-objective optimization concepts [35], [169], etc. Compared with these two techniques, repair algorithms have the advantage of directly harnessing domain knowledge, e.g., identifying non-critical constituent services for downgrading cloud resources, for handling the budgetary constraint of the *CAD* problem.

2.2.2 Local Search

Local search is a heuristic method for solving combinatorial optimization problems [84]. At each iteration, local search finds an improved solution by searching the *neighborhood* of the current solution until a solution deemed optimal is found or a time-bound is elapsed [4]. Local search can be trapped in local optima in the search space that are better than all their neighbors, but not necessarily representing the best possible solution, i.e., the global optimum. To improve the effectiveness of local search, various algorithms have been introduced. For example, Simulated Annealing (SA) [185], Tabu Search (TS) [56], Variable Neighborhood Search (VNS) [65], Large Neighborhood Search (LNS) [149], and Guided Local Search (GLS) [191] all attempt to help local search escape local optimum. These local search algorithms are widely applied to challenging combinatorial optimization problems, including the traveling salesman problem [190] and nurse scheduling problem [1].

Large Neighbourhood Search (LNS)

The LNS-based algorithms in [68] have successfully solved the problem of service brokering in multi-cloud. As a combination of local search and constraint programming, LNS is proposed by Shaw [149] to take the advantages of both *exploration* and *propagation*. LNS makes moves as local search but uses a tree-based search with constraint propagation to evaluate the cost and legality of the move. The moves can bring substantial changes by changing a large portion of the solution. The potential for changing large parts of the solution gives LNS its name. Typically the neighbourhood's size varies exponentially with the number of basic elements of the solution changeable by the move [101].

In [68], LNS is designed to make large changes to the current solution by continually using a destroy heuristic and a repair heuristic. After one iteration of destruction and repair, the new solution is evaluated to de-

Algorithm 2 LNS approach [68]

Input: The maximum iterations $iter_{max}$

Initialize a solution S
$S_{best} \leftarrow S, i \leftarrow 0$
while $i < iter_{max}$ **do**
  $S' \leftarrow destroy(S)$
  $S' \leftarrow repair(S')$
  if S' is better than S_{best} **then**
    $S \leftarrow S'$
    $S_{best} \leftarrow S$
    $i \leftarrow 0$
  else
    $i \leftarrow i + 1$
  end if
end while

Output: S_{best}

termine whether to reject or accept as the current solution (see Algorithm 2). Due to its flexibility in designing problem-specific destroy and repair heuristics, LNS is promising to address the *ARD with flexible dispatching* problem that involves complex optimization decisions.

2.2.3 Reinforcement Learning

Inspired by behavioural psychology, Reinforcement Learning (RL) [170] is the area of machine learning that deals with sequential decision-making. Recently, many RL approaches have been proposed for resource management problems in the literature [29], [31], [100], [105], [122], [175], [215].

RL problem usually can be formalized as an agent that needs to make decisions in its environment to optimize cumulative rewards shown in Figure 2.2. At each time step t, the agent observes the state s_t and choose

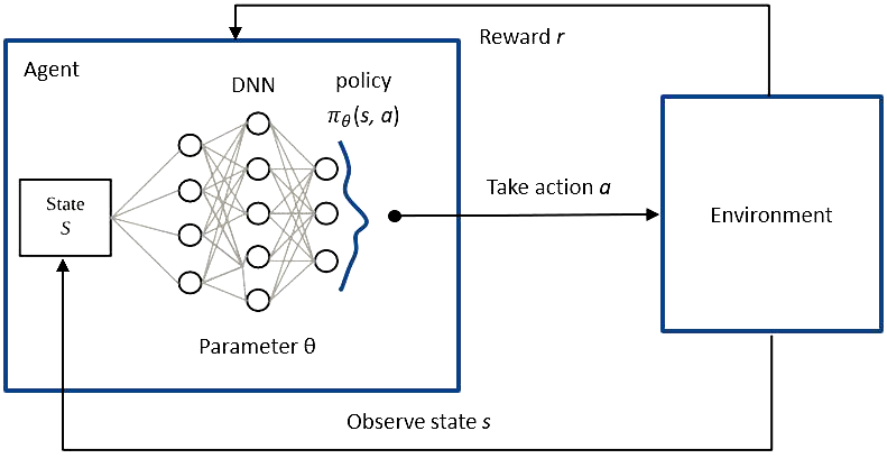


Figure 2.2: RL with policy represented via DNN.

an action a_t. Following the action, the state transitions to s_{t+1} and the agent obtains reward r_t. The system is assumed to have Markov property, i.e., the probabilities of state transition and the rewards only depend on the state s_t and the corresponding action a_t.

It is worth noting that the agent has no prior knowledge about which state will be transitioned or how much reward will be received. Only during the training process, the agent can observe previous transitions by interacting with the environment. The goal of the agent is to learn an optimal, or nearly-optimal, action selection policy to maximize the expected cumulative reward.

There are two main approaches for RL, i.e., *value-function*-based methods and *policy-search*-based methods [11]. Specifically, the value-function-based methods estimate the value (i.e., expected return) in a given state and choose the best action in the state. The policy-search-based methods aim to directly search for an optimal policy π^*. Also, there are hybrid actor-critic methods combining both the value function and policy search [117].

Deep Reinforcement Learning (DRL)

DRL is the combination of RL and deep learning [12]. DRL has been able to solve a wide range of complex decision-making problems with high dimensional state-space by learning different levels of abstractions from data, e.g., video games [118], Computer Go [162], etc. DRL applies *Deep Neural Networks* (DNNs) [62] as function approximators shown in Figure 2.2.

DQN is one of the well-known value-function-based DRL algorithms and is the first RL algorithm that was demonstrated to work directly from raw visual inputs in many learning environments [118]. Concretely, the input to the DQN is the state. The state is processed by several convolutional and fully connected layers with ReLU rectifiers between each layer. At the final layer, the network outputs Q-value. A discrete action is further selected based on the Q-value. Then the game returns a new score given the current state and chosen action. The difference between the new score and the previous one, that is, the Temporal-Difference (TD) error is used to learn from the decision.

DQN applies Q-learning [194], a model-free RL algorithm, to learn the value of an action in any state. Concretely, Q-learning updates the Q-function by:

$$Q^{new}(s_t, a_t) \leftarrow (1 - \alpha) \cdot Q(s_t, a_t) + \alpha \cdot (r_t + \gamma \cdot \max_a Q(s_t, a)) \tag{2.1}$$

where α is the learning rate ($0 < \alpha \leq 1$) and γ is the discount factor ($0 < \gamma \leq 1$).

The strength of DQN is its ability to compactly process both the high-dimensional visual inputs (as the state) using DNN [11]. Besides, experience replay [96] and target networks [118], are employed by DQN to address the fundamental instability problem from using function approximation in RL [183].

Safe RL

For some problems, it is important to ensure reasonable system performance (e.g., performance constraints of the *EAD* problem) and/or safety of the agent (e.g., expensive robotic platforms) during the learning [50]. Safe RL tries to learn a policy that maximizes the expected return, while also ensuring (or encouraging) the satisfaction of the safety constraints [30]. In the literature, there are two main approaches for safe RL: (1) transforming the optimization criterion (e.g., to *constrained criterion*) [50], and (2) modifying the exploration process (e.g., by *safe exploration*) [49]. Next, we briefly introduce the two main approaches.

The constrained criterion is applied to constrained Markov processes in which we need to maximize the expected return while keeping other types of expected utilities lower or higher than given bounds. This approach transforms constrained problems into unconstrained problems by various methods, e.g., the penalty methods [166].

There are two ways of modifying the exploration process to avoid risky situations: (1) through the incorporation of external knowledge, and (2) through the use of a risk-directed exploration [50]. In the former case, prior knowledge can be incorporated into the exploration process by (1) providing initial knowledge [41], (2) deriving a policy from a finite set of demonstrations [2], and (3) providing teacher advice [49]. In the latter case, a risk measure is used to determine the probability of selecting different actions during the exploration process while remaining the optimization criterion [51].

2.2.4 Long Short-Term Memory

Recently, machine learning techniques have introduced new methods to time series analysis. In particular, deep learning methods are capable of identifying the structure and pattern of data such as non-linearity and complexity in time series analysis [161].

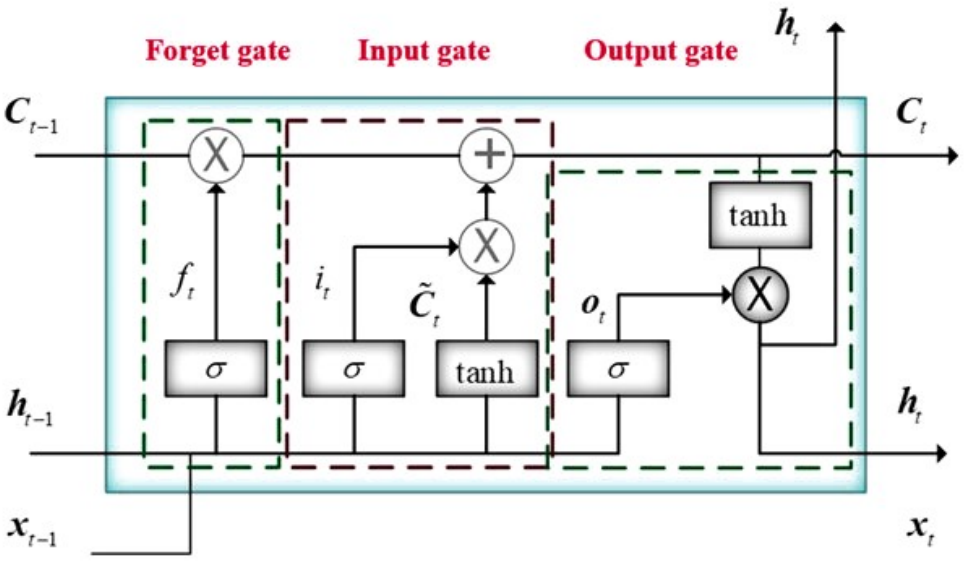


Figure 2.3: A LSTM unit.

Long Short-Term Memory (LSTM) is a special case of the Recurrent Neural Network (RNN) method that was initially introduced by Hochreiter and Schmidhuber [70]. A LSTM unit is composed of a cell state and three types of gates: *forget gate*, *input gate*, and *output gate* shown in Figure 2.3. The cell state stores data coming from gates and the gates regulate the flow of information into and out of the cell state. Concretely, forget gate conditionally decides what information to throw away from the unit, the input gate conditionally decides which values from the input to update the cell state, and the output gate conditionally decides what to output based on input and the cell state. These gates have weights that are learned during the training procedure. By overcoming the vanishing gradient problem, LSTM is capable to predict application workloads over time with excellent performance [79], [91].

RL and LSTM have been respectively applied to automatically scale containers with promising results in [79], [141], [213]. In this thesis, we will develop a DRL-based approach with an LSTM-based workload prediction

model and safe mechanisms to address the *EAD* problem.

2.3 Related Work

Related work discusses the problem formulation and existing approaches for application deployment in cloud. The problems of *CAD*, *ARD*, and *EAD* are reviewed in Subsection 2.3.1, Subsection 2.3.2, and Subsection 2.3.3, respectively.

2.3.1 Composite Application Deployment

This subsection introduces the related work about the *CAD* problem. The main challenges that need to be addressed are also highlighted in this subsection.

Service Deployment for Composite Applications

In recent years, cloud computing is becoming a booming hosting paradigm for delivering enterprise applications. How to deploy application services with optimal QoS becomes a critical issue [107], [195]. Several research works have studied the service deployment problem [107], [153], [195]. For example, Wen et al. studied application deployment on federated clouds to minimize the monetary cost while simultaneously meeting the security and reliability requirements in [195]. The work in [107] proposed a game-theoretic method to optimize both the overall cost and execution time of cloud services. From the perspective of application providers, however, application deployment is often subject to stringent budgetary control to ensure financial viability.

Based on Service-Oriented Architecture (SOA) [28], there is an ongoing trend to deploy composite applications in the form of workflows with shared services [75], especially for enterprise application providers. In [75], Huang et al. described the inter-dependence and independence among

services as a Service Relationship Graph (SRG). The problem was then solved by transforming it into a minimum k-cut problem and deploying a set of services onto the given VMs. In this thesis, we focus on deploying composite applications in multi-cloud where there are various VM types available at different data centers.

Because enterprise applications potentially involve a large number of user requests and different applications must cater to varied user distributions, the processing of ongoing application requests can be modeled through a queuing model. Although the queuing theory has been well studied to analyze the performance of cloud-based systems [89], [203], existing research works on cloud-based application deployment did not pay full attention to its role in performance evaluation based on the average response time [47], including the request processing time and the network latency between users and the deployed application.

Focusing on cloud service selection alone, the application deployment problem in multi-cloud is similar to the multi-cloud service brokering problem, which has also received some research attention [68], [152], [181]. In [68], [152], the service brokering problem was explored to cope with not only the deployment cost but also the network latency between users and cloud data centers around the world, in view of the significant impact of network latency on the performance of cloud services. However, different from the *CAD* problem with inter-dependent services in this thesis, the service brokering problem in [68], [152] assumes all services as being independent, which may not be always true, especially when services are used to fulfil specific functionalities in workflows.

The review of the above-mentioned works reveals a need for the study on the *CAD* problem considering the budgetary constraint and average response time of composite applications with inter-dependent services.

Budget-Constrained Workflow Scheduling

Selecting proper cloud resources for processing a workflow of tasks is another commonly studied problem, which is also known as the workflow scheduling problem [165], [199], [219]. Most of the existing works focus on scheduling *scientific* workflows with data-intensive or computation-intensive tasks [99]. Compared with the time-consuming execution of these tasks and data transfer between tasks, the network latency between users and cloud resources is negligible. Some research on workflow scheduling explicitly recognized the budget control as the key constraint [145], [208], [209], [216]. Motivated by strong commercial concerns, budgetary control is much more important for deploying enterprise applications [25].

As an extension of the well-known Heterogeneous Earliest Finish Time (HEFT) algorithm [180], Budget-constrained Heterogeneous Earliest Finish Time (BHEFT) was proposed in [216] for budget-constrained workflow scheduling. BHEFT first calculates the priorities of all tasks in a workflow according to their average execution time and data transfer time among different resources then allocates the total budget for each task with regard to spare application budget, current task budget, and adjustment factor in order of the priority. If no adequate solution that satisfies the budget constraint can be discovered, the scheduling is considered as a failure.

Another category of approaches for budget-constrained workflow scheduling is based on the back-tracking heuristic, e.g., [145], [209]. These algorithms firstly generate an initial schedule to minimize the execution time. Then the chosen tasks are gradually migrated to downgraded resources in an attempt to meet the budgetary requirement. However, these search-based strategies can easily get trapped in local optima and exhibit inferior performance [9].

Meta-heuristic methods also have been used for budget-constrained workflow scheduling, e.g., the GA-based approach in [208]. To deal with the budget constraint, Yu et al. [208] applied a penalized fitness function to evolve the budget-compliant schedule. Similarly, a co-evolutionary

GA with adaptive penalty function CGA^2 was proposed in [99] to handle deadline-constrained workflow scheduling, which minimizes total cost while meeting the deadline. The crossover and mutation rates were co-evolved in CGA^2 to help convergence.

In addition to penalty functions, many other constraint-handling techniques have been proposed for GAs, such as repair algorithms [36], [110]. Particularly, the repair algorithms aim to transform an infeasible solution into a feasible one during the evolutionary process. The problem-dependent repair algorithms have been widely demonstrated with a strong ability to handle constraints both effectively and efficiently on a diverse range of constrained optimization problems [36]. However, the computation time required for repairing the solution is usually higher than other techniques [146]. Therefore, an efficient repair algorithm is needed to control the repair time. For example, the repair upper bound according to the proportion of feasible individuals in the current population can be dynamically adapted to improve the efficiency of repair algorithms.

2.3.2 Application Replication and Deployment

This subsection introduces the related work about the *ARD* problem. We also highlight the main challenges in this subsection.

Service-oriented Replication in Cloud

Replication techniques are commonly exploited to ensure high availability and satisfactory QoS [164]. There are two major ways to achieve replication in cloud, i.e., replicating the data and replicating the service. They are termed as Data-oriented Replication (DoR) and Service-oriented Replication (SoR) [26]. Compared with DoR, SoR consumes not only the storage resource but also other resources such as CPU, memory, network, bandwidth, etc [164].

Recently, SoR has attracted much attention from researchers [119], [164].

For example, some works replicate services in several locations to ensure the availability of the provisioned services [21], [127], [200]. Some works focus on reducing energy consumption for cloud providers by adjusting the number of replicas according to the workload, e.g., [19]. From the perspective of application providers, there are some works minimizing the deployment cost of replicas [20], [54], [182]. However, these works do not consider the average response time, which seriously affects the Quality of Experience (QoE) of applications among geographically distributed users [104], [201].

Because the *ARD* problem deploys application replicas in multi-cloud, how to dispatch user requests among all application replicas has an important impact on the average response time of applications. In the literature and industry, dispatching user requests to the closest application replicas is a common practice [113], [171], [201]. Therefore, we need to study the *ARD with close dispatching* problem subject to the constraints on the average response time.

Note that the close dispatching approach may result in deployed VMs under-utilized. To further reduce the deployment cost, the user requests from the same region can be dispatched to different application replicas. Therefore, we should study the *ARD with flexible dispatching* problem. The *ARD with flexible dispatching* problem introduces extra optimization decisions, i.e., how to dispatch user requests among all application replicas, which makes *ARD* more complicated.

Data Placement and Replication in Cloud

The cloud-hosted data placement and replication problems have received much attention in the recent literature. Integer Linear Programming (ILP) has been considered as the dominant method to model these problems [66], [201]. For example, Pyramid was proposed to maximize both the utility- and locality-awareness of replicas for P2P cloud storage systems in [66]. ILP was implemented to find the placement of replicas. To serve the

demands on videos, the problem of finding the optimal content deployment and request dispatch strategy was formulated as Mixed-Integer Linear Programming (MILP) in [201]. The efficient solutions were achieved by using dual decomposition [204] and linear programming techniques. These ILP problems consistently minimize network latency as the optimization objective. In this thesis, we consider the response time of user requests, including both the network latency and the request processing time. For the cloud-based applications under high workloads, the request processing time is nonlinearly related to the capacity of deployed VMs and users' demands [47]. The nonlinear nature renders these ILP-based approaches inapplicable for the *ARD* problem.

Many meta-heuristic algorithms have been used for the data placement and replication problem in cloud [88], [102]. In [102], two GA-based approaches were proposed to replicate data objects over multiple sites to avert undesired long delays experienced by end-users. To optimize social media data placement and replication in cloud data centers, a GA-based approach was presented to minimize monetary cost while satisfying latency requirements for all users in [88]. However, these approaches focus on the location selection for data replicas, while the *ARD* problem must decide both the locations and the types of cloud resources. Besides, these approaches do not explicitly dispatch widely distributed user requests among all application replicas, which is a critical component of the *ARD with flexible dispatching* problem.

To sum up, the *ARD with close dispatching* problem is challenging due to its nonlinear nature and large search space. For the *ARD with flexible dispatching* problem, we also need to design effective methods to optimize the dispatching of application requests based on the corresponding replica deployment so that the total deployment cost is minimized while satisfying the constraint on average response time.

2.3.3 Elastic Application Deployment

This subsection introduces the related work about the *EAD* problem. The main challenges that need to be addressed are highlighted.

Container Deployment in Multi-cloud

Currently, elastic development and run-time management of applications increasingly rely on containers, an industry-leading lightweight virtualization technology [18], [140]. Significant efforts have been made for container deployment in multi-cloud. The commercial container management platforms, e.g., Rancher [138] and OpenShift [67], facilitate container deployment in multi-cloud. In addition to multi-cloud management and visibility, they empower applications with providers the ability to easily adapt the deployment of containers across multi-cloud data centers through a unified user interface or API. However, these platforms do not support fully automated deployment and adaptation for applications to respond to the workload fluctuations, which is essential for application providers to achieve low deployment cost and satisfactory application performance [6].

The problem of container scaling in cloud has been studied at different resource levels: container deployment [5], [141], [177], cluster deployment [40], [217], and both [167]. These works use horizontal scaling [40], [177], [217], vertical scaling [5], or both [141] depending on different elasticity dimensions. However, they focus on auto-scaling techniques within a single data center to handle workload variations of applications in a cost-effective manner. That is, these solutions lack the ability to decide container locations in their adaptation processes, which is essential to reduce network latency of applications through the lightweight and portable nature of containers. Considering the locations of containers, the existing algorithms for the deployment of containerized applications (e.g., using meta-heuristics in [6]) suffer a high computational cost. Moreover, these

algorithms usually do not consider the impact of current deployment decisions on the future deployment cost and average response time.

Resource Allocation with Reinforcement Learning

RL has been proposed to support VM scaling in [10], [17], [29], [82] because it can ensure stable resource utilization of applications when the workload changes dynamically [213]. Due to the large difference between containers and VMs in terms of start-up, shut-down, and migration times, these RL-based algorithms that perform well in VM scaling cannot be applied to container scaling effectively [213]. Recently, some RL-based algorithms have been proposed for scaling containerized applications. For example, Horovitz et al. [72] proposed an RL-based approach to dynamically adapt the thresholds used to horizontally scale the containers. Rossi et al. [141] proposed a model-based RL method to control the horizontal and vertical elasticity of containers. Because the future trend of workload is not considered, these approaches may fail to deal with highly dynamic application workload, which causes resource waste or unsatisfactory QoS [213].

As a prediction technique, time series analysis has been applied to handle the dynamic application workloads [23]. The strategy has been used for RL methods by combing workload prediction models to ensure the timeliness and accuracy of scaling actions, such as A-SARSA [213]. Again, A-SARSA focuses on auto-scaling containers within a single data center. When considering both various configurations and locations of containers in multi-cloud, the Q-table used in A-SARSA cannot handle the high-dimension state-space effectively.

In recent years, DQN was applied in cloud-based resource allocation problems with high-dimension state space [29], [100], [207]. In [29], a DQN-based resource provisioning and task scheduling system was proposed to minimize the energy cost for cloud service providers. Further considering the thermal effect of job allocation, Yi et al. [207] applied DQN to allocate compute-intensive jobs within the boundary of a single

data center. Another work in [100] proposed a two-level VM allocation and power management framework. On the one hand, DQN was used at the global tier for allocating VMs to hosts. On the other hand, RL and workload analysis were used in the local tier to manage the power consumption.

The works in deep RL assume that agents are free to explore any behaviour during learning [93], [95], [117], [118], [148], [162]. For the *EAD* problem, it is unacceptable to give an agent complete freedom. For example, some scaling actions could cause containerized applications to be heavily utilized. In that case, the application performance will drastically deteriorate so that the constraint on average response time cannot be satisfied. Therefore, safe exploration for RL agents is important [8], [120]. To guarantee constraint satisfaction, Constrained Policy Optimization (CPO) is proposed to train neural network policies for high-dimensional control [3]. However, CPO is inapplicable for the *EAD* problem because it requires the learning process to start from a feasible policy, which may not be easy to find.

The review of the above-mentioned works motivates the DQN-based algorithm to address the *EAD* problem. Moreover, to satisfy the constraint of the *EAD* problem on average response time, safe RL techniques, e.g, penalty-based reward function and problem-tailored safe exploration, can be applied.

2.3.4 Datasets

This subsection introduces the important datasets that have been normally used in the related work.

- Composite applications: In [75], ten composite applications of eight workflow structures are considered to simulate different enterprise services such as online shopping and travel planning. The processing time of constituent services for a single request is measured on

the VMs from Amazon EC2.

- Application requests distribution: In [198], the distribution of application requests is based on Facebook subscribers. The latest statistics can be found in World Population Review[1].

- Multi-cloud resources: In [68], real datasets of multi-cloud resources are collected from global leading cloud providers, such as AWS and Microsoft. Different VM types in several regions of respective cloud providers are considered with real pricing schemes.

- Network latency: Sprint IP backbone network databases[2] provide a real-time snapshot of IP network performance among eighty-two global locations.

In conclusion, the above datasets are from different sources. Because there are no benchmark datasets for multi-cloud application deployment on the global scale, we will synthetically utilize these datasets in this thesis.

2.4 Summary and Thesis Scope

This chapter introduced the main concepts of application deployment in multi-cloud, optimization and machine learning methods and also reviews the related work. We discussed the limitations of existing works on the three application deployment scenarios, i.e., *CAD*, *ARD*, and *EAD*, in multi-cloud. The limitations and challenges for location-aware application deployment in multi-cloud were summarized in the following four aspects.

- The current problem formulations of cloud-based application deployment lack consideration of the location of cloud resources, which

[1]https://worldpopulationreview.com/country-rankings/facebook-users-by-country
[2]https://www.sprint.net/tools/ip-network-performance

is important for multi-cloud application deployment from the perspective of application providers with geographically distributed users. Therefore, new problem formulations that include the aspect of location need to be defined.

- GA is a promising technique for the *CAD* problem with an extremely large search space. The difficulties of developing GA methods for the *CAD* problem are the location-related search space and efficient constraint handling. The dependency-based service clustering and self-adaptive repair algorithm are potentially suitable for the *CAD* problem. Their effectiveness needs to be studied in-depth.

- The *ARD* problem considers service-oriented replication in multi-cloud, which introduces extra optimization objectives, e.g., the number of replicas. We should design a new optimization approach under the GA framework to address *ARD* with a common request dispatching approach in practice, i.e., the *ARD with close dispatching* problem. We also study the *ARD with flexible dispatching* problem, which needs to optimize the deployment of application replicas and the dispatching of user requests simultaneously. A new two-stage optimization framework combining MILP and LNS should be developed to ensure the effectiveness and efficiency of the proposed approach.

- CaaS is widely used in practice for the *EAD* scenario. In multi-cloud, the horizontal scaling involves containers from different cloud providers with different prices at different locations. In the literature, RL is a promising technique for auto-scaling cloud resources for applications with dynamic workloads. To minimize the deployment cost while satisfying the constraint on average response time, the prediction technique based on time series analysis and safe RL should be investigated for DRL to solve the *EAD* problem.

Table 2.1: The thesis scope

Characteristic	Thesis scope
Perspective	Application providers
Application type	Interactive application
Service Model	IaaS and CaaS
Interconnected environment	Multi-cloud
Objective	Average response time (*CAD*) Deployment cost (*ARD* and *EAD*)
Constraint	Budget (*CAD*) Average response time (*ARD* and *EAD*)
Major methods	GA, LNS, LSTM, and DRL

The thesis scope is shown in Table 2.1. We investigate various approaches to help *application providers* deploy *interactive applications* in *multi-cloud*. The thesis considers *IaaS* and *CaaS* for different scenarios, i.e., *CAD*, *ARD*, and *EAD*. We further consider *average response time (CAD)* and *deployment cost (ARD and EAD)* as objectives and *budget (CAD)* and *average response time (ARD and EAD)* as constraints. Our literature review successfully identified several important optimization and machine learning techniques, including *GA, LNS, LSTM, and DRL*, that will be utilized to solve these constrained optimization problems in this thesis.

Chapter 3

Composite Application Deployment

3.1 Introduction

This chapter studies Composite Application Deployment (*CAD*) in multi-cloud. In practice, it is typical for an application provider to offer several business applications simultaneously, and the relevant business processes are usually abstracted as workflows [218]. These composite applications consist of a set of specific functionalities, i.e., constituent services. Different applications often share some common constituent services [75]. For example, a tax-setting service can be shared by ordering, selling, and payroll applications.

Under the public cloud paradigm, many deployment patterns, such as IaaS, PaaS, and FaaS, are available for application services. Application providers often prefer IaaS in practice in order to achieve a high level of control and flexible management goals [81]. For *CAD* in multi-cloud, we consider the major scenario of deploying application services on a set of VMs to support composite applications capable of processing tens of thousands or even millions of requests on a daily basis [205].

The deployment of constituent services is crucial since it affects the

Quality of Service (QoS) of composite applications significantly [75], [192]. The problem has received some research attention in recent years. For example, the service deployment problem is solved by minimizing the *execution time* of composite applications without attention to the locations of users and data centers in [75]. In multi-cloud, however, the *network latency* between users/data and services in different locations may significantly affect the application performance [68], [152]. Therefore, to evaluate accurately the performance of composite applications, we must consider the *average response time*, measured from the moment users make application requests to the moment when they receive the corresponding responses. Driven by the goal to minimize the average response time, we investigate the *CAD* problem by selecting proper VMs in multi-cloud to jointly support multiple composite applications subject to diverse user distributions and quality requirements.

It is widely known that leading cloud providers usually provide cloud resources, e.g., VM instances, with varied pricing subject to regions[1]. In consideration of both performance and deployment cost, there are some research works searching for a set of deployment solutions, in which each solution represents a unique trade-off deployment plan [171], [219], or minimizing the product of deployment cost and response time of applications [153]. However, enterprises are often more interested in budgetary control to ensure that their actual costs adhere closely to their financial plan [25]. In view of this demand, the budget management module is offered by many cloud providers, e.g., Amazon Web Service (AWS)[2]. The budgetary control emphasizes performance optimization within a given budget, which is of immense practical significance for application providers [142]. In this thesis, we formulate the *CAD* problem as a *budget-constrained* optimization problem so as to truly meet the business management needs of many enterprises.

[1]https://aws.amazon.com/ec2/pricing/on-demand/

[2]https://aws.amazon.com/aws-cost-management/aws-budgets/

GA is a classic optimization technique to generate high-quality solutions for various combinatorial optimization problems with practical importance [39], [197]. In particular, driven by a population-based solution improvement framework, GA has been widely applied to multi-cloud resource allocation for scalable cloud application deployment [69], [85], [211]. However, new algorithmic techniques under the GA framework are required to address the *CAD* problem. First, a large number of different resources are provided at multi-cloud data centers. The large and complex search space can impair the scalability and applicability of the existing GA-based approaches [61]. Second, numerous research works in the literature have shown that repair algorithms are powerful in handling constrained optimization problems by problem-specific heuristics [36], [146]. We need to propose a specific repair algorithm to solve our problem effectively because there are no standard guidelines for the design of repair algorithms [111]. Third, to control the computation time of the repair process, we should also consider the efficiency of our proposed repair algorithm [146].

To address each of the challenges above, we have developed new algorithmic techniques under the GA framework for effective *CAD*. First, we propose a service clustering mechanism based on the dependency among services to reduce the complexity/size of the search space, thereby ensuring the efficiency and effectiveness of our GA-based algorithm. Second, we propose a problem-specific repair algorithm to progressively downgrade the service hosting VM types so as to meet the budget requirements with low-performance deterioration. Third, the repair algorithm is designed to achieve a desirable trade-off between performance and computation cost by transforming the over-budget solutions chosen according to an adaptive bound into constraint-compliant solutions. The main contributions of this chapter are summarized as follows.

Firstly, we formally define the *CAD* problem in multi-cloud as a constrained optimization problem with the goal to minimize the response

time of composite applications subject to a budget constraint. To the best of our knowledge, this is the first study in the literature on the application deployment problem with considerations of both the application performance and the budget impact in multi-cloud at the global scale.

Secondly, we propose a hybrid GA-based approach, namely *H-GA*, to the *CAD* problem, featuring a newly designed and domain-tailored service clustering method, repair algorithm, solution representation, population initialization, and genetic operators.

Finally, to evaluate the proposed approach, we collect the information about VM capacity and pricing offered by the global top three cloud providers, i.e., Amazon, Microsoft, and Alibaba. Based on the collected information, extensive experiments have been conducted to deploy a large group of composite applications with different service diversities and budget factors. Our approach is compared to several state-of-the-art algorithms, i.e., BHEFT [216] and penalty-based GAs [99], [208] for constrained application deployment problems. The experimental results indicate that our approach significantly outperforms the existing methods, achieving more than 8% performance improvement in terms of response time, and 100% budget satisfaction in the meantime.

3.2 Chapter Organization

The remainder of this chapter is organized as follows. Section 3.3 presents the *CAD* problem formulation. Section 3.4 introduces the hybrid GA-based approach. The service clustering mechanism, chromosome representation, fitness evaluation, population initialization, repair algorithm, and genetic operators are respectively described in this section. To evaluate the proposed approach, we conduct a series of experiments in Section 3.5. Section 3.6 summarizes this chapter.

3.3 *CAD* Problem Formulation

The deployment of composite applications is achieved by selecting a collection of hosting VMs in multi-cloud to minimize the application response time within a given budget. In this section, we first define composite applications and introduce multi-cloud application deployment systems, then formulate the *CAD* problem in multi-cloud. The key notations to be used for problem definition are listed in Table 3.1.

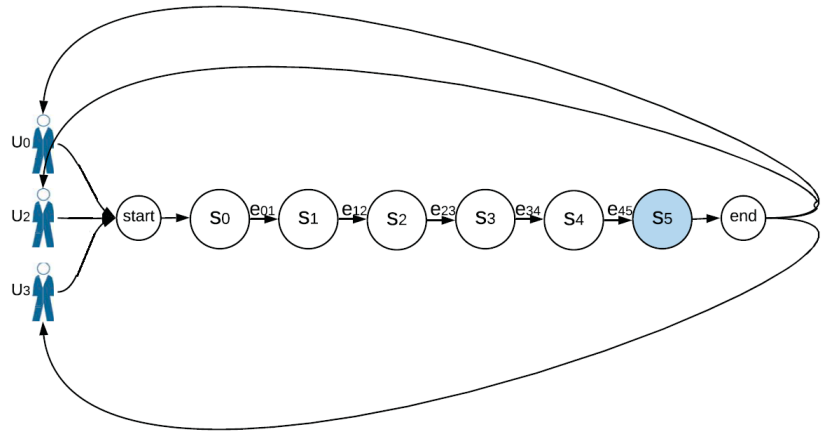
3.3.1 Composite Application

An application provider provides a set of composite applications $\mathcal{A} = \{A_0, ..., A_a, ..., A_{|\mathcal{A}|-1}\}$ that jointly use a collection of constituent services $\mathcal{S} = \{s_0, ..., s_s, ..., s_{|\mathcal{S}|-1}\}$. We represent each application A_a as a Directed Acyclic Graph (DAG), denoted by $G(S_a, E_a)$. $S_a \subseteq \mathcal{S}$ is a set of nodes representing services. Every service s_s might be shared by multiple composite applications. E_a is a set of directed edges where $e_{mn} \in E_a$ connects s_m with s_n. Based on e_{mn}, s_m is called the parent service of s_n and s_n is the child service of s_m. In application A_a, service $s_s \in S_a$ may have more than one parent services and child services, we denote the set of the parent services of s_s as $Pre(s_s)$, and the set of the child services of s_s as $Succ(s_s)$. If a service s_s requires access to any dataset during processing, we denote this service as $\dot{s_s}$. Each application A_a has exactly one dummy starting service that has no parent services, and one dummy ending service that has no child services, which are denoted as $start_a$ and end_a respectively. Three representative composite application examples from [75] are shown in Figure 3.1. In the example, service s_1 is shared by applications A_0 and A_1, while $\dot{s_5}$ involves all three applications and is also a data-access service (represented in blue circles).

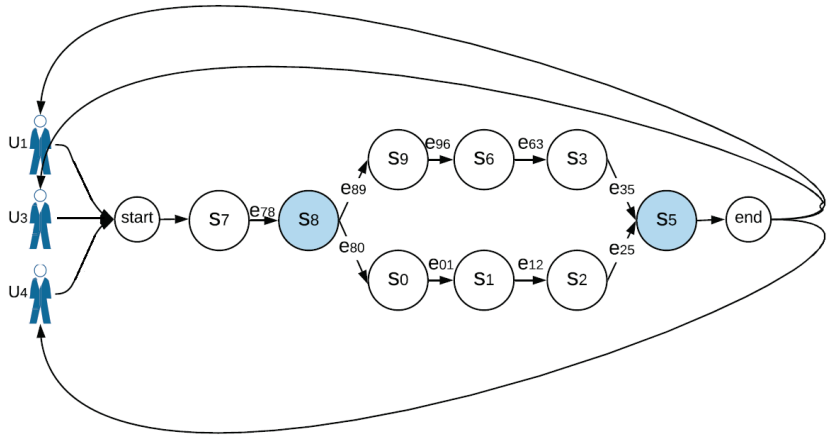
We refer to [171] and consider a set of user centers $\mathcal{U} = \{U_0, ..., U_u, ..., U_{|\mathcal{U}|-1}\}$, which represent the centers of the global user groups. The request rates change periodically, e.g., on an hourly basis. Within each hour, the

Table 3.1: Mathematical notations for the *CAD* problem

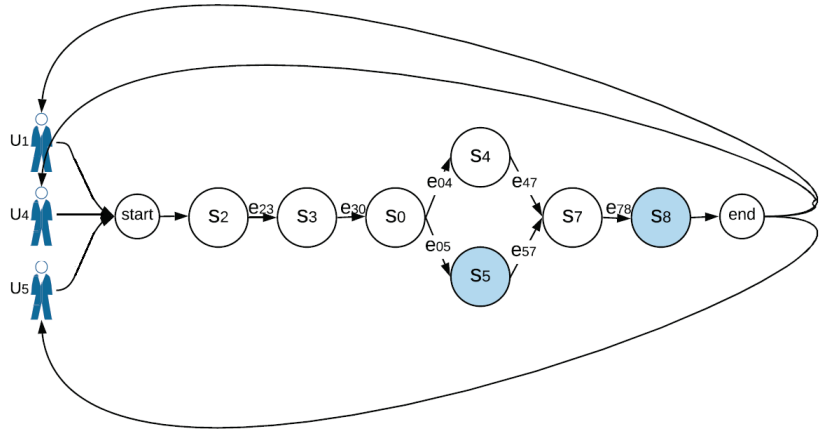
Notation	Definition
A_a	The a^{th} composite application
s_s	The s^{th} constituent service in composite applications
S_a	The set of services in the a^{th} composite application
E_a	The set of directed edges in the a^{th} composite application
e_{mn}	The directed edge from the m^{th} service to the n^{th} service
$start_a$	The starting service of application A_a
end_a	The ending service of application A_a
U_u	The u^{th} user center
V_v	The v^{th} type of VM
C_c	The c^{th} data center
v_c	The VM instance that belongs to the v^{th} type of VM in the c^{th} data center
rc_{vc}	The rental cost of v_c
γ_{au}	The request rate of A_a from the u^{th} user center
θ_a	The workload of application A_a
β_s	The workload of constituent service s_s
α_s	Processing capacity of the s^{th} service
pt_s	The average processing time of the s^{th} service
dt_s	Data access delay of the s^{th} service
It_{au}	Network latency from the u^{th} user center to starting service of the a^{th} application
Ot_{au}	Network latency from ending service of the a^{th} application to the u^{th} user center
ut_a	Average network latency between user centers and the a^{th} application
st_{mn}	Network latency from s_m to s_n
b	The specified budget by the application provider
k	The budget factor
is_a	The time required for inter-service communication within application A_a



(a) Composite application A_0



(b) Composite application A_1



(c) Composite application A_2

Figure 3.1: Three example composite applications for deployment.

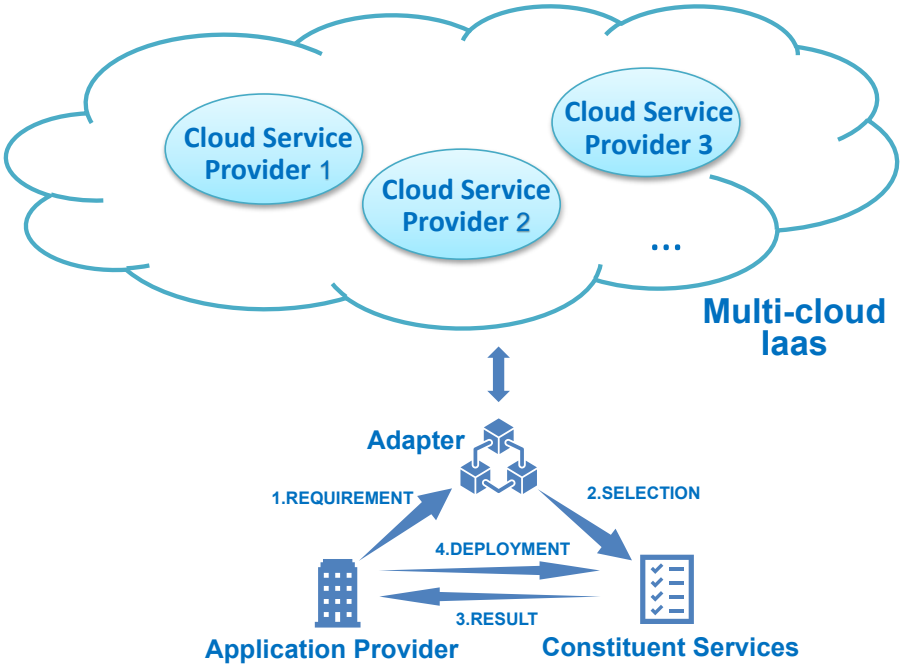


Figure 3.2: A *CAD* system in multi-cloud based on [178].

request rate γ_{au} denotes the number of requests from a user center U_u to the starting service of the application A_a per time unit. In this regard, the total workload of A_a can be determined as:

$$\theta_a = \sum_{u=0}^{|\mathcal{U}|-1} \gamma_{au}. \tag{3.1}$$

3.3.2 Multi-cloud Application Deployment System

Application providers create their composite applications by composing a set of constituent services with specific functionalities. These services can be deployed onto a set of VMs in multi-cloud (shown in Figure 3.2).

We consider a set of VM types $\mathcal{V} = \{V_0, ..., V_v, ..., V_{|\mathcal{V}|-1}\}$ and a set of multi-cloud data centers $\mathcal{C} = \{C_0, ..., C_c, ..., C_{|\mathcal{C}|-1}\}$. If VM type V_v is available in data center C_c, we use v_c to denote the instance of VM type V_v and

rc_{cv} to denote its rental cost per time unit.

We assume each constituent service is deployed to one VM instance. This is a common assumption in the literature [75], [195] and widely exercised in the industry [163]. We further assume that one VM instance is only allocated to one constituent service in order to minimize interferences between any two services. Each service instance maintains an independent queue on a First Come First Served (FCFS) basis for pending requests arriving from all user centers or its parent services in all deployed applications. Following [75], the average resource consumption per request over a long sequence of requests for the same service is considered highly stable, while different services can incur different resource requirements. We follow [187] and model the operation of each individual service instance as an $M/M/1$ queue [7], [124]. Because a constituent service can be shared by multiple applications, the aggregated workload of service s_s can be calculated by:

$$\beta_s = \sum_{a=0}^{|\mathcal{A}|-1} \theta_a \qquad \text{s.t.} \quad s_s \in S_a. \tag{3.2}$$

For data-access service $\dot{s_s}$, we use dt_s to denote the data access time between service $\dot{s_s}$ and the closest dataset deployed either by the application provider or a third party. We assume that request processing starts from data access, i.e., once the requested data becomes available, the service deployed to a VM instance will process the data. Considering the data access, the adjusted capacity of a VM instance with capacity α_s can be computed by:

$$\widetilde{\alpha_s} = \frac{1}{\frac{1}{\alpha_s} + dt_s}. \tag{3.3}$$

According to Little's Law [98], the average request processing time of service s_s is:

$$pt_s = \begin{cases} \frac{1}{\widetilde{\alpha_s}-\beta_s} & \text{if } \dot{s}_s \\ \frac{1}{\alpha_s-\beta_s} & \text{otherwise.} \end{cases} \tag{3.4}$$

We do not take the data access cost into account in the overall budget constraint, because the cost is usually negligible compared with the service deployment cost[3]. We also have an operation constraint, i.e., $\alpha_s(\widetilde{\alpha_s}) > \beta_s$, that guarantees that the workload of the VM instance will not exceed its capacity, i.e., each constituent service is deployed to a capacity-feasible VM.

We define the average Response Time of an application RT_a as the total time on average required for A_a to process all user requests. Since a user center $U_u \in \mathcal{U}$ and a data center $C_c \in \mathcal{C}$ can span large geographic areas, we must take network latency into account when computing RT_a. The *average* network latency ut_a between user centers and the application A_a can be determined by:

$$ut_a = \frac{\sum_{u=0}^{|\mathcal{U}|-1} \gamma_{au}(It_{au} + Ot_{au})}{\theta_a}, \tag{3.5}$$

where It_{au} is the network latency between a user center U_u and the starting service of A_a, while Ot_{au} is the network latency between the ending service of A_a and user centers U_u. Besides, we denote the network latency between service s_m and its child service s_n as st_{mn}. The aforementioned three types of network latency, i.e., dt_s, ut_a, and st_{mn}, can be treated as constants in the communication network, which are usually determined by several non-changeable factors such as the physical distance among datasets, user centers, and services. Accordingly, the RT_a can be calculated as follows:

$$RT_a = ut_a + MS(A_a), \tag{3.6}$$

where function MS calculates the response time of A_a (also called *makespan*). We define two functions EST (Earliest Start Time) and FT (Finish Time)

[3]https://azure.microsoft.com/en-us/pricing/details/bandwidth/

to calculate the response time. The computing process begins with the starting service, and MS is the finish time of the ending service:

$$\begin{aligned} &EST(start_a) = 0, \\ &FT(s_s) = EST(s_s) + pt_s, \\ &EST(s_s) = \max_{s_m \in Pre(s_s)} \{FT(s_m) + st_{ms}\}, \\ &MS(A_a) = FT(end_a). \end{aligned} \tag{3.7}$$

For example, based on the above formulas, we can compute RT of A_2 in Figure 3.1 as follow:

$$\begin{aligned} RT_2 =& ut_2 + MS(A_2) \\ =& ut_2 + FT(end_2) \\ =& ut_2 + EST(s_8) + pt_8 \\ =& ut_2 + FT(s_7) + st_{78} + pt_8 \\ =& ut_2 + EST(s_7) + pt_7 + st_{78} + pt_8 \\ =& ut_2 + \max\{(FT(s_4) + st_{47}), (FT(s_5) + st_{57})\} + pt_7 + st_{78} + pt_8 \\ & \ldots \\ =& ut_2 + pt_2 + st_{23} + pt_3 + st_{30} + pt_0 \\ & + \max\{(st_{04} + pt_4 + st_{47}), (st_{05} + pt_5 + st_{57})\} + pt_7 + st_{78} + pt_8. \end{aligned}$$

3.3.3 *CAD* Problem

We define a service deployment vector $X = [x_{v,c,s}]_{s=0}^{|\mathcal{S}|-1}$ to represent the application deployment solution, where $x_{v,c,s}$ indicates that a service s_s is deployed to one instance of VM type V_v in the data center C_c. That is, the deployment cost of s_s, i.e., DC_s, is rc_{cv}. The total Deployment Cost (TDC) and Average Response Time (ART) of all composite applications can be calculated as follows:

$$TDC = \sum_{s=0}^{|\mathcal{S}|-1} DC_s \tag{3.8}$$

$$ART = \frac{\sum_{a=0}^{|\mathcal{A}|-1} \theta_a \cdot RT_a}{\sum_{a=0}^{|\mathcal{A}|-1} \theta_a}. \tag{3.9}$$

Therefore, the *CAD* problem aims to minimize ART of all the composite applications, as defined in Formula (3.9), subject to the budgetary constraint:

$$\begin{aligned} \min_{X} \quad & ART \\ \text{subject to} \quad & TDC \leq b, \end{aligned} \tag{3.10}$$

where b is the budget specified by the application provider. We use *budget feasible* or *feasible* solutions to refer to solutions satisfying budgetary constraints in the remaining of this chapter. We refer to [145] and define b in eq. (3.11):

$$b = C_{cheapest} + k \cdot (TC_{highest} - TC_{cheapest}), \tag{3.11}$$

where $TC_{highest}$ and $TC_{cheapest}$ are the total costs of deployments by using either the most expensive VM instance or the cheapest capacity feasible VM instances for all the services respectively. $k \in [0, 1]$ is the budget factor to determine the tightness of budgetary control. The larger k is, the more budget the application provider has.

In summary, different from the existing problem formulations for *CAD*, we consider the locations of cloud resources as optimization decisions, adopt the queuing model to precisely capture the dynamics involved in the processing of ongoing application requests, and formulate the *CAD* problem as a budget-constrained optimization problem for enterprise application deployment.

3.4 A hybrid GA-based Approach: *H-GA*

In multi-cloud, there are a large number of different VM types available at different data centers, resulting in extremely large search spaces (theoretical size is $(|\mathcal{V}| \cdot |\mathcal{C}|)^{|\mathcal{S}|}$). We first propose a clustering method to group relevant services and deploy the services in the same cluster to the same data center. This clustering technique can substantially reduce the size of the search space. In recent years, repair algorithms have become relatively promising solutions to handle constraints in GAs [36]. These techniques demonstrate promising performance for various combinatorial optimization problems, thanks to problem-specific heuristics [146]. However, when using repair algorithms to solve constrained optimization problems, computation time is one of the critical issues. Consequently, we propose a repair algorithm to downgrade the non-critical services for the over-budget solutions within an adaptive bound to improve efficiency without hurting performance.

Figure 3.3 shows the overall process of the proposed Hybrid GA-based approach (*H-GA* for convenience). After service clustering and encoding, an initial population is generated. In each generation, our algorithm first repairs the population within an adaptive upper bound, then evaluates the fitness of the repaired solutions. Finally, the problem-specific selection, crossover, and mutation operations are performed to generate a new population. The population is evolved iteratively until the predefined maximum number of generations is reached. Finally, the best chromosome is decoded to derive the final application deployment solution.

In the following subsections, we will introduce in detail the service clustering, encoding/decoding scheme, fitness evaluation, problem-specific population initialization, repair algorithm, and genetic operators.

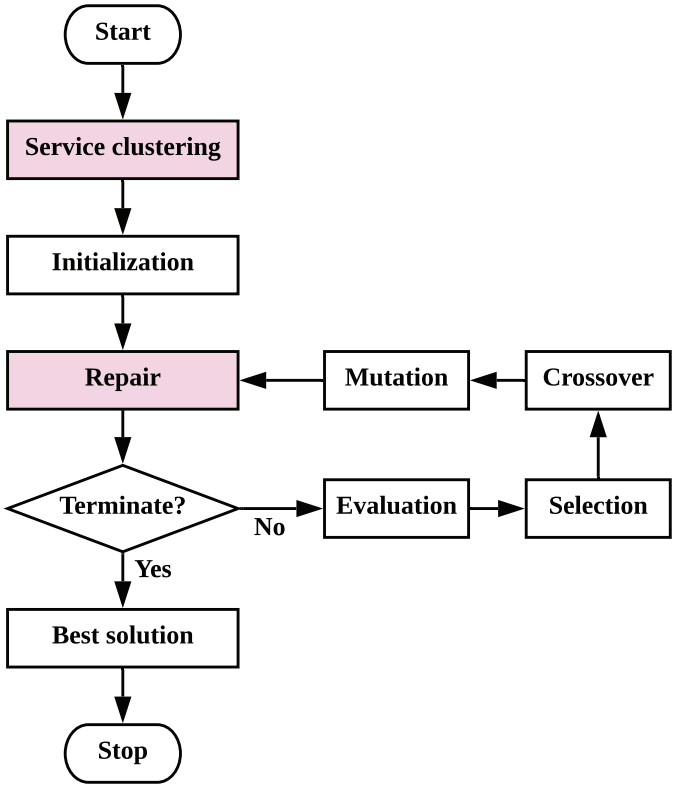


Figure 3.3: The framework of *H-GA* with highlighted algorithmic techniques.

3.4.1 Service Clustering

Regarding the VM location selection problem, if the constituent services are considered separately, the size of the search space is $|\mathcal{C}|^{|\mathcal{S}|}$ (maximum value is 15^{40} in our experiments). It is essential to reduce the search space to guarantee algorithm scalability. Inspired by [143] that recommends the execution of pipeline tasks on the same data center to save data transfer time, we decide to cluster *dependent services* and deploy the services in the same cluster to the same data center to reduce network latency. A straightforward method to determine the dependency of a service on any

Algorithm 3 Service clustering algorithm

Input: $\mathcal{A}$, $\mathcal{S}$

Output: clustered services $\mathcal{SC}$

1: **for all** $s_s \in \mathcal{S}$ **do**
2: Find the application(s) with the highest request rate among applications that include s_s, i.e., $\mathcal{A}_s$
3: **if** $|\mathcal{A}_s| = 1$ **then**
4: Assign s_s to the cluster c_a, where $A_a \in \mathcal{A}_s$
5: **else**
6: Find the application $A_a \in \mathcal{A}_s$ with the most services and assign s_s to the cluster c_a
7: **end if**
8: **end for**

application is to compare the request rates experienced by the same service from different applications.

In practice, we regard each application as the centroid of a separate service cluster. Concretely, if a service is shared by multiple applications, we assign it to the cluster c_a where application A_a has the highest request rate. Hence, we can guarantee that the services in the same cluster will communicate more frequently than services from different clusters. For the service that has an equal request rate with respect to more than one applications, we simply assign it to the cluster with the highest number of services. By doing so, the communication among a larger number of services can be easily localized to the same data center, reducing the networking overhead.

After service clustering, the size of the search space is reduced to $|\mathcal{C}|^{|\mathcal{A}|}$. In other words, the size of search space after service clustering is $|\mathcal{C}|^{|\mathcal{S}|-|\mathcal{A}|}$ times less than that of the original problem without clustering. More details about the clustering algorithm can be found in Algorithm 3.

Assume that the request rates of the three applications in Figure 3.1 are

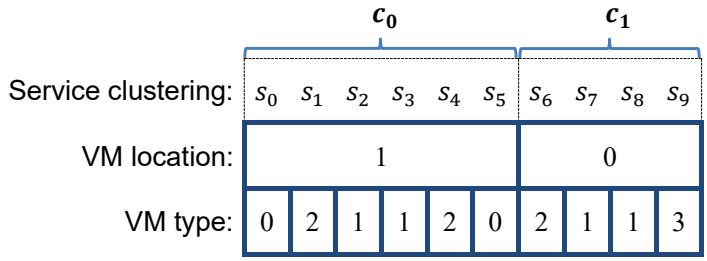


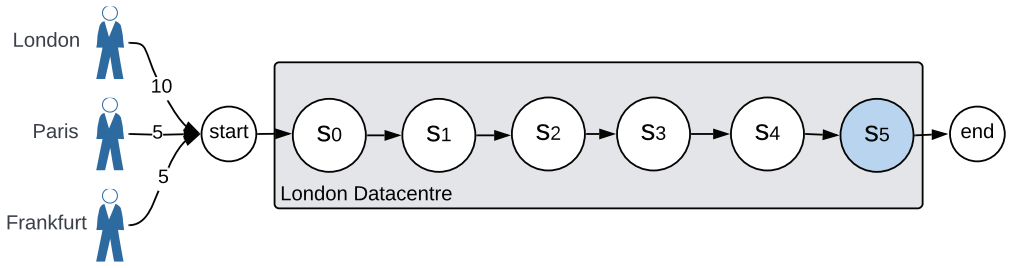
Figure 3.4: An example chromosome for encoding deployment solutions.

20, 15, and 10 per second, the clustering algorithm will group 10 services into two clusters, i.e., c_0 and c_1 as demonstrated in Figure 3.4.

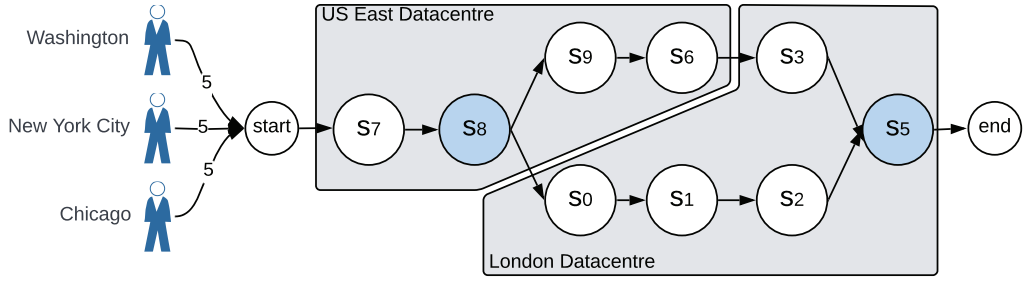
It is important to note that service clustering can effectively reduce network latency across all applications. Refer to the example shown in Figure 3.1. For application A_0, all its services in c_0 will be deployed to the same data center, e.g., AWS London shown in Figure 3.5(a). Let is_a denote the time required for inter-service communication within A_a. In this case, we assume that the inter-service communication time of A_0, i.e., is_0, is negligible. According to the application request distribution in Figure 3.5(a), the average network latency between users and A_0 can be calculated according to eq. (3.5):

$$\begin{aligned} ut_0 &= \frac{\sum_{u=0}^{|\mathcal{U}|-1} \gamma_{0u}(It_{0u} + Ot_{0u})}{\theta_0} \\ &= \frac{10 \cdot (2+2) + 5 \cdot (7+7) + 5 \cdot (17+17)}{20} \\ &= 14(ms). \end{aligned}$$

On the other hand, for other applications such as A_1 with its services grouped into more than one cluster, we can still effectively control the inter-service network latency. In particular, if all the users of A_1 are in America as shown in Figure 3.5(b), and we still allocate cluster c_0 to AWS London but cluster c_1 to AWS US East, then the average network latency



(a) Composite application A_0



(b) Composite application A_1

Figure 3.5: A VM location deployment example with request distribution.

between users and W_1 can be calculated by:

$$
\begin{aligned}
ut_1 &= \frac{\sum_{u=0}^{|\mathcal{U}|-1} \gamma_{1u}(It_{1u} + Ot_{1u})}{\theta_1} \\
&= \frac{5 \cdot (12+92) + 5 \cdot (12+92) + 5 \cdot (8+88)}{15} \\
&\doteq 101.33(ms)
\end{aligned}
$$

The inter-service network latency is_1 is about 80ms (from US East to London). This is clearly better than the alternative situation if we allocate all services of A_1 to AWS US East. In that case, we can ignore the inter-service network latency of A_0 and A_1, i.e., $\Delta is_0 = is'_0 - is_0 = 0ms$ and $\Delta is_1 = is'_1 - is_1 = -80ms$, however, the average network latency between

users and the two applications will change to:

$$\begin{aligned} ut'_0 &= \frac{10 \cdot (82+82) + 5 \cdot (87+87) + 5 \cdot (97+97)}{20} \\ &= 174(ms) \\ ut'_1 &= \frac{5 \cdot (12+12) + 5 \cdot (12+12) + 5 \cdot (8+8)}{15} \\ &\doteq 21.33(ms). \end{aligned}$$

That is, $\Delta ut_0 = 160ms$ and $\Delta ut_1 = -80ms$, and $\Delta ART_0 = \Delta ut_0 + \Delta is_0 = 160ms$ and $\Delta ART_1 = \Delta ut_1 + \Delta is_1 = -160ms$. Note that application A_0 is much more frequently requested by end users than application A_1. This results in longer ART than the original deployment solution according to eq. (3.9).

3.4.2 Chromosome Representation

We use chromosomes to encode service deployment solutions. We split a chromosome into two strings, i.e., *VM location* and *VM type*, to represent the mappings from services to data centers and VM types respectively. Based on the clustering result, the services in the same cluster are treated as a single deployment unit in any *VM location* string.

Figure 3.4 gives the encoding of a CAD solution shown in Figure 3.1. This encoded solution (aka. individual in the GA population) gives a complete deployment solution, according to which service s_0 is deployed to a VM instance of type V_0 at data center C_1, s_1 is deployed to the VM instance of type V_2 at the same data center C_1, and so on.

3.4.3 Fitness Evaluation

For each chromosome, we evaluate its fitness by calculating ART based on eq. (3.9). We also calculate TDC to determine the necessity of applying the repair process to enforce budgetary control. Our proposed repair algorithm is described in Subsection 3.4.5.

3.4.4 Initial Population

Generally, GAs generate the initial population randomly to ensure its diversity. To improve the solution quality and convergence speed, a portion of the initial population will be determined based on a straightforward greedy heuristic (VM locations selection) and the BHEFT [216] (VM types selection).

For the *VM location* string, we apply a greedy method to find the data center location with minimal network latency between the users and service clusters. The greedy heuristic is presented in Algorithm 4. For each candidate data center location, we calculate the network latency between it and all user centers (step 7). If there are data access services in this cluster, the data access delay is also added (step 12). Finally, the location of each service cluster with the minimal network latency is chosen to form the initial *VM location* string for subsequent evolution. In Figure 3.4, for example, two clusters are deployed in data centers C_1 and C_0 respectively.

Next, we initialize the VM types based on BHEFT [216], which is an extension of the well-known DAG scheduling heuristic HEFT [180] by considering the budget constraint. BHEFT first distributes the total budget to tasks (constituent service in our problem) in the order of their priorities, then selects appropriate VMs for tasks with regards to their distributive budgets.

For *each* composite application, we follow BHEFT to recursively calculate the rank of every service in this application, as defined by:

$$rank_s = \overline{pt_s} + \max_{s_m \in Succ(s_s)} (st_{sm} + rank_m) \tag{3.12}$$

where $\overline{pt_s}$ is the average processing time for s_s on all capacity feasible VM types in the selected data center and st_{sm} is the network latency from s_s to s_m based on the selected data centers. The calculation begins with the ending service whose rank equals 0 (dummy service).

Because the services are shared by multiple composite applications, for

Algorithm 4 Greedy heuristic for VM locations selection

Input: $\mathcal{C}$, $\mathcal{A}$, $\mathcal{U}$, γ_{au}, It_{au}, Ot_{au}, service cluster set $\mathcal{SC}$

Output: service cluster deployment location

```
for all service cluster ∈ SC do
  for all data center location C_c do
    latency_c ← 0
    for all s_s in cluster c do
      if s_s connects to starting service or ending service of A_a directly then
        for all U_u ∈ U do
          latency_c ← latency_c + It_au(Ot_au) · γ_au
        end for
      end if
      if s_s requires data access during processing then
        for all A_a include s_s do
          latency_c ← latency_c + dt_s · θ_a
        end for
      end if
    end for
  end for
  Return the data center location with the minimal latency
end for
```

each service we sum up all its rank values from corresponding applications as its comprehensive priority. Then we deploy the services in the order of priority. The best possible VM type is selected for each service based on Spare Application Budget (SAB) and Current Service Budget (CSB) as follows:

$$\begin{aligned} SAB_s &= b - \sum_{i=0}^{s-1} DC_i - \sum_{j=l}^{|\mathcal{S}|-1} \overline{DC_j} \\ CSB_s &= \overline{DC_s} + SAB_s \cdot AF_s, \end{aligned} \tag{3.13}$$

where DC_i is the cost of the deployed service s_i, and $\overline{DC_j}$ is the average cost for the undeployed service s_j on all its capacity feasible VM types. The Adjustment Factor (AF) acts as a weight to tune the impact of SAB_s on CSB_s. Based on [216], AF is defined as the ratio between the average cost of the current service to the sum of the average costs of the undeployed services as follows:

$$AF_s = \begin{cases} \frac{\overline{DC_s}}{\sum_{j=s}^{|\mathcal{S}|-1} \overline{DC_j}} & \text{if } SAB_s \geq 0 \\ 0 & \text{otherwise.} \end{cases} \tag{3.14}$$

The rest of the population will be initialized randomly. Meanwhile, we ensure that services in the same cluster are located in the same data center. In each data center, services are deployed to *available* and capacity feasible VM types randomly.

3.4.5 Repair Algorithm

The solutions randomly generated initially and evolved subsequently may violate the given budget constraint. Therefore, to make the infeasible solutions constraint-compliant we use a repair algorithm. In particular, our repair operator aims to reduce the Total Deployment Cost (TDC) step by step to be within the given budget.

To reduce the TDC of the given set of applications, for some of their services we can either select the same type of VM but at a different data center with a cheaper price or select a cheaper VM at the same data center. For example, we can change the location of service from London to US East, which offers cheaper services than in London, or downgrade the VM type to a cheaper one, e.g., from AWS m5.large to m5.samll. Based on the coding scheme introduced in Subsection 3.4.2, however, the change of VM locations will affect all the services in the same cluster and may cause a huge fluctuation of ART. Consequently, we design our repair algorithm to downgrade VM types for some selected services while simultaneously evolving VM location selection through GA.

Our repair algorithm is based on the concept of *critical-path* [180], which determines the overall response time of an application. Inspired by the work in [180], which proposes critical-path-based heuristics to optimize the makespan of workflows, we propose to downgrade VM types for services that are not on any critical path of composite applications to reduce TDC.

We use *non-critical service* to denote a service that is not on any critical path of the given composite application. To reduce the overall cost without increasing ART, our repair algorithm first identifies non-critical services and then downgrades VM types with a lower but feasible capacity to save operational cost. For example, for a non-critical service deployed to AWS m5.xlarge, our repair algorithm can change the VM type to m5.large, without or with little impact on ART. By identifying and downgrading non-critical services iteratively, the repair algorithm improves over-budget solutions until the TDC falls within the budget.

Note that repairing infeasible application deployment solutions can be computationally expensive due to a number of possible ways to repair, and therefore impacts the overall efficiency of *H-GA*. Replacing infeasible solutions with existing feasible solutions is an efficient strategy. However, this approach can decrease the diversity of the GA population and therefore affect the algorithm's effectiveness. Therefore, we should repair a sufficient number of infeasible solutions to maintain the diversity of the populations of GA. To achieve a desirable trade-off between computation time and algorithm effectiveness, we set an adaptive upper bound $\widehat{TC}$ on TDC. With $\widehat{TC}$ changing dynamically during the evolution process, only the newly evolved solutions with its TDC falling in between a given budget b and $\widehat{TC}$ will be repaired. The repair algorithm is described in Algorithm 5.

Refer to [176], the adaptive $\widehat{TC}$ is calculated as follows:

Algorithm 5 Repair algorithm

Input: current population $\mathcal{P}$, b, $\mathcal{V}$, $\mathcal{S}$

Output: repaired population $\mathcal{P}'$

1: Calculate $\widehat{TC}$ based on eq. (3.15) and determine the best feasible solution $best$ which has the minimal ART
2: **for all** $ind \in \mathcal{P}$ **do**
3: **if** $TDC_{ind} > \widehat{TC}$ **then**
4: Replace ind with $best$
5: **else**
6: **while** $TDC_{ind} > b$ **do**
7: **for all** $s_s \in \mathcal{S}$ **do**
8: Identify the non-critical service and downgrade its VM type with a lower but feasible capacity
9: **end for**
10: **end while**
11: Return the repaired $\mathcal{P}'$
12: **end if**
13: **end for**

$$\begin{aligned} \widehat{TC} &= b + (TC_{max} - b) \cdot (1 - r_f) \\ r_f &= \frac{N_f}{N}, \end{aligned} \tag{3.15}$$

where TC_{max} is the maximum cost with respect to any solution in the current population, N_f is the number of feasible solutions in the current population, and N is the population size. In this way, $\widehat{TC}$ decreases whenever the proportion of feasible solutions r_f is increased. In that case, fewer infeasible solutions will be repaired so as to save the repair time.

If we use the probability density f_{TDC} to illustrate the distribution of the TDC among all solutions in the population, r_f can be presented by the cumulative distribution function $F_{TDC}(x)$ as $F(b)$. Similarly, the pro-

portion of feasible solutions after repair can be denoted as $F(\widehat{TC})$. We assume that f_X is monotonically decreasing within the interval $[b, TC_{max}]$, therefore $F(b)$ is convex between b and TC_{max}. Then we can easily get:

$$\frac{F(\widehat{TC}) - F(b)}{\widehat{TC} - b} \geq \frac{F(TC_{max}) - F(b)}{TC_{max} - b},$$

where $F(C_{max}) = 1$. According to eq. (3.15), we can conclude as follows:

$$\begin{aligned} F(\widehat{TC}) - F(b) &\geq \frac{1 - F(b)}{TC_{max} - b} \cdot (TC_{max} - b) \cdot (1 - F(b)) \\ F(\widehat{TC}) - F(b) &\geq (1 - F(b))^2 \\ F(\widehat{TC}) &\geq (F(b) - 0.5)^2 + 0.75, \end{aligned}$$

where $F(\widehat{TC}) \in [0.75, 1]$. That is, the repair algorithm with adaptive bound will guarantee at least 75% feasible solutions in the GA population. The proportion can usually ensure good performance in GAs [63]. For the solutions with a cost higher than C_u in each generation, we simply replace them with the best feasible solution in terms of ART in the current population.

3.4.6 Genetic Operators

We apply the roulette-wheel selection to choose individuals for breeding. This selection method is the most commonly used selection method for GAs [97]. In this section, we present our problem-specific crossover operator and mutation operator as follows.

Crossover

The crossover operator is designed differently for VM location and VM type. For VM location, each *cluster* in new *VM location* strings is chosen from parents with equal probability. For VM type, each *digit* in new *VM type* strings is chosen from parents with equal probability. This uniform

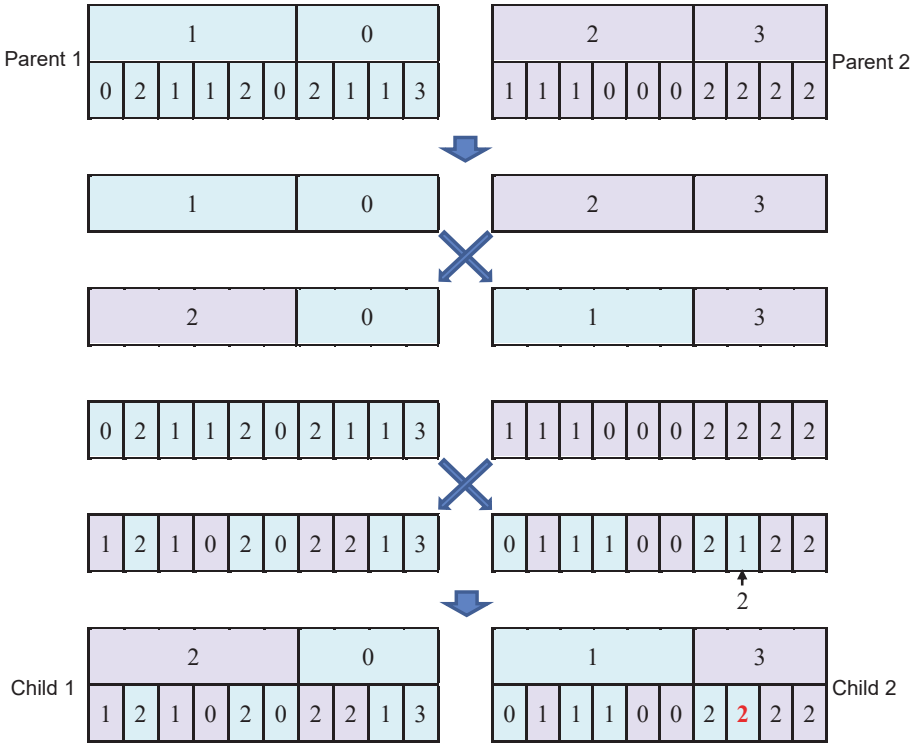


Figure 3.6: An example of the crossover operator.

crossover can be seen as a generalization of the more traditional one- and two-point crossover operators [33]. An example of this operation is given in Figure 3.6, in which the new *VM location* strings and *VM type* strings from two crossover operations are combined into new chromosomes. Note that the combination may generate an *unavailable solution* with unavailable VM types in the new data center. In this case, we modify the solutions to be available by replacing the unavailable VM type with the most similar available VM type. For example, the available V_2 in C_3 (in red) is chosen if V_1 is unavailable in C_3 in Figure 3.6.

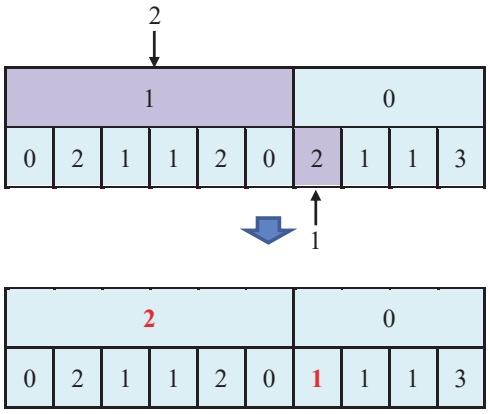


Figure 3.7: An example of the mutation operator.

Mutation

The mutation operator can introduce a local improvement to previously evolved solutions. Similar to the crossover operator, our mutation operator mutates the strings of *VM location* and *VM type* separately. Firstly, the mutation operator randomly selects a service cluster in the *VM location* string and randomly replaces it with a different data center. Secondly, a digit in the *VM type* string is selected randomly and updated with a different and capacity feasible VM type. Finally, the availability of all VM types in the new location is checked and modified in a similar fashion as described in the crossover operator. Figure 3.6 demonstrates an example where the first cluster and service s_6 are randomly chosen and replaced.

3.5 Evaluation

In the absence of a publicly available global multi-cloud testbed, we conduct a series of experimental studies to examine the performance of *H-GA* by comparing it with three state-of-the-art algorithms for constrained ap-

plication deployment, i.e., *BHEFT* [216], the Penalty-based GA (*P-GA* for convenience) [208], and *CGA*^{2} [99].

P-GA applies a penalized fitness function to evolve the budget-compliant solutions. Concretely, the fitness function is formulated as $\frac{ART}{maxTime}$ if the cost of an application deployment solution is within the given budget, where $maxTime$ is the biggest ART of the current population. Otherwise, *P-GA* defines its fitness function as $1 + TDC/b$. That is, when the cost of an application deployment solution exceeds the given budget, the fitness of the solution is penalized in terms of a cost-fitness TDC/b.

CGA^2 is proposed to handle deadline-constrained workflow scheduling, which minimizes total cost while meeting the deadline. The crossover and mutation rates were co-evolved in CGA^2 to help convergence. To adapt *CGA*^{2} to our problem, we change its cost-related fitness value to ART and time-related constraint violation to TDC. For a fair comparison, both *P-GA* and *CGA*^{2} apply our proposed problem-specific chromosome representation and genetic operators.

3.5.1 Datasets

Based on the latest report regarding the worldwide IaaS public cloud services market share [168], we collect the real VM type descriptions and pricing schemes in February 2021 from three leading cloud providers, i.e., AWS[4], Microsoft Azure[5], and Alibaba Elastic Compute Service (ECS)[6]. 24 different VM types (8 from AWS, 8 from Azure, and 8 from Alibaba) have been included in the experiments. We also consider a total of 15 locations for major AWS, Azure, and Alibaba data centers, i.e., Northern Virginia, Northern California, Dublin, London, Paris, Frankfurt, Stockholm, Hong Kong, Singapore, Seoul, Tokyo, Sydney, Sao Paulo, Mumbai, and Montreal. 8 locations for hosting application-dependent datasets are Northern

[4]https://aws.amazon.com/ec2/instance-types/
[5]https://azure.microsoft.com/en-us/services/virtual-machines/
[6]https://www.alibabacloud.com/product/ecs

California, Hong Kong, Sao Paulo, Mumbai, London, Tokyo, Frankfurt, and Sydney, which are their main data regions. Furthermore, we adopt 82 user centers from 35 countries on 6 continents in the Sprint IP Network[7] to simulate the global user community.

To determine the network latency between users/data and deployed services, and the network latency from parent services to their respective child services, we follow real-world observations of network latency from Sprint IP backbone network databases[8].

3.5.2 Experiment Settings

In our experiments, we consider eight business workflow structures corresponding to diverse activities such as online shopping and travel planning services based on [75] (see Figure 3.8). The range of service processing time for a single request is also obtained from [75], that is, 5-35 ms running on AWS t2.micro. Three scales of services, i.e., 20 (low), 30 (medium), and 40 (high) different services, are implemented to simulate the different business diversities. For different service diversities, we generate 8 composite applications based on randomly selected workflow structures and services.

Referring to [198], we apply Facebook subscribers statistics[9] to simulate the distribution of application requests among all user centers. By January 2020, there are approximately 1.3 billion Facebook subscribers from all of the 35 countries considered in our simulation. We consider the total number of users for our simulated Web applications as 1/1000th of Facebook subscribers to simulate the application providers with millions of users[10]. That is, the number of users in different regions ranges from 2100 to 71800. For each composite application, we select 30 out of all 82 user

[7]https://www.sprint.net

[8]https://www.sprint.net/tools/ip-network-performance

[9]https://worldpopulationreview.com/country-rankings/facebook-users-by-country

[10]https://github.com/qingdaost/LBARDM

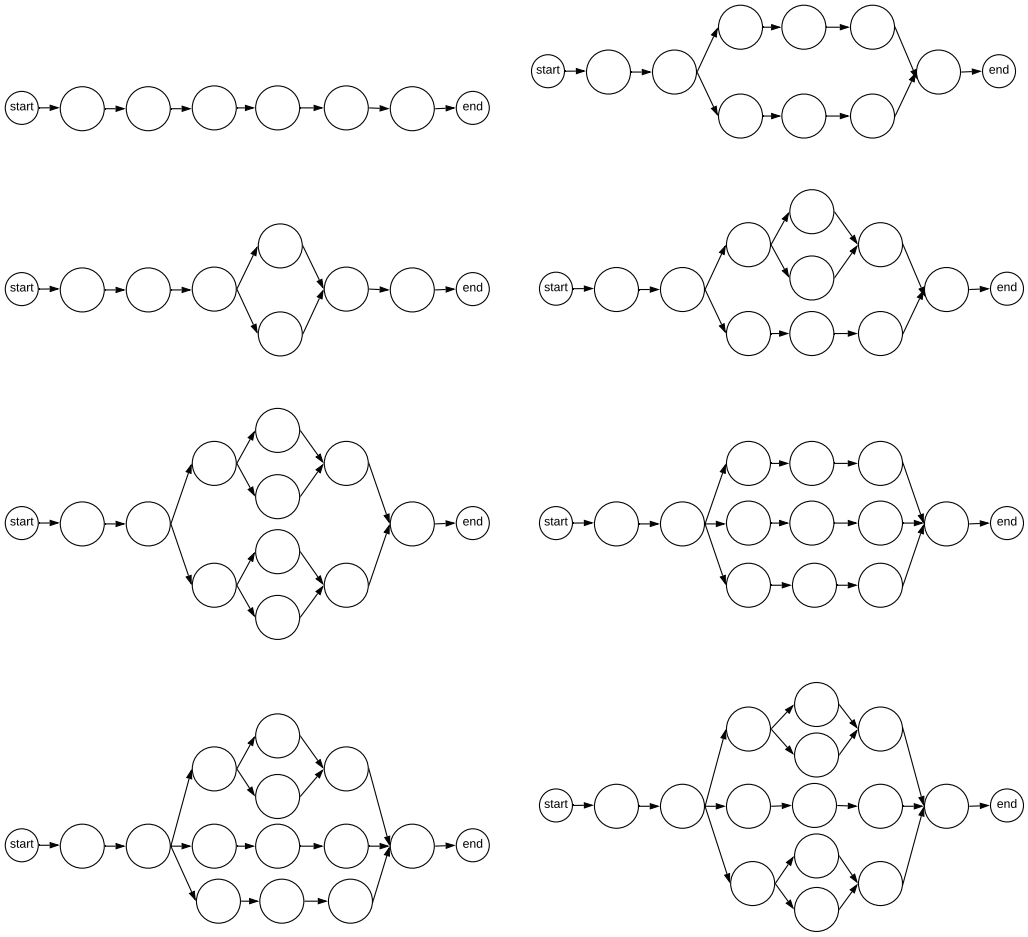


Figure 3.8: The business application structures used in the experiments.

centers randomly, and each user from the selected user centers makes 25 requests on average daily. Accordingly, the application request rate spans from 52 to 304 requests per second (that is, approximately 4.5-26.3 million requests daily). We assume that new application requests from user centers are generated according to a Poisson distribution, following the common practice in many previous works [15], [60]. Refer to [9], we implement 6 budget factors, i.e., 0.1, 0.2, 0.3, 0.4, 0.7, and 0.9 for each set of composite applications.

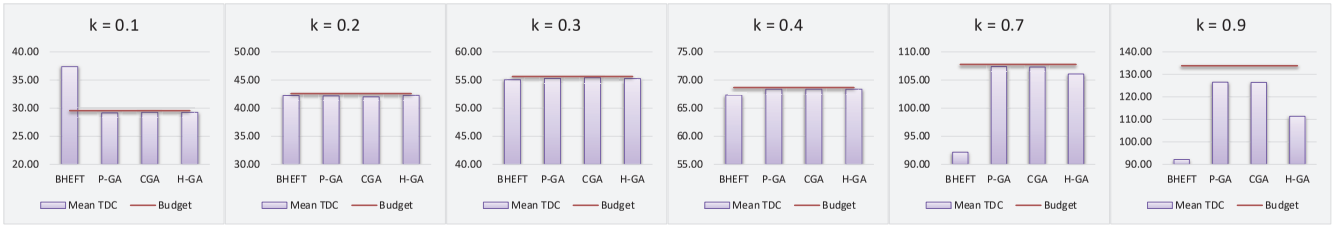
3.5.3 Algorithm Parameter Settings

Because the *BHEFT* algorithm [216] does not support VM location selection, we choose VM instances from the most economical data center to save cost. For *P-GA* approach, our parameter settings include: population size $N = 1024$, maximum generation $G = 1000$, which are sufficient to perform well. That is, we cannot gain significant improvements on the quality of the final solution by using more generations and larger populations. For *CGA*^{2}, we implement the same parameters with [99]: $M_1 = 200, G_1 = 100$ for searching deployment solutions, and $M_2 = 50, G_2 = 20$ for searching suitable crossover and mutation rates. For *H-GA* approach, our parameter settings include: population size $N = 100$, maximum generation $G = 1000$. Following [99], the proportion of greedy and *BHEFT* based solutions in the initial population is 20%. The crossover rate r_c and mutation rate r_m of *P-GA* and *H-GA* are 0.9 and 0.1 respectively following common practice in the literature [32]. To compare the results, we consider the mean and standard deviation of ART and TDC after running each experiment 30 times. All the experiments are performed on the same computer with Intel Core i7-8700 CPU (3.2 GHz and 16 GB of RAM).

3.5.4 TDC Evaluation

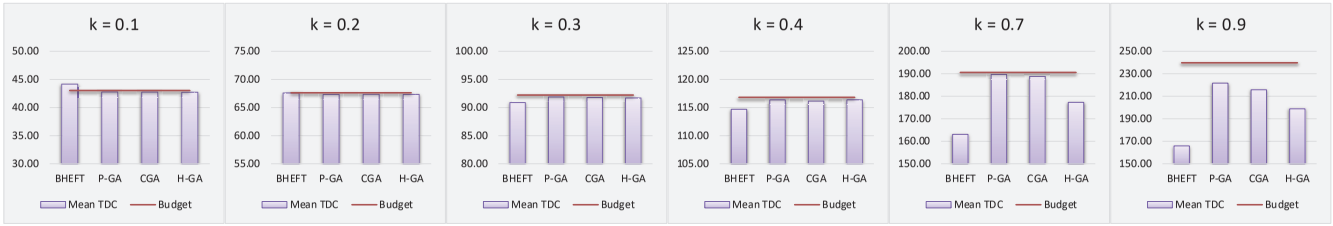
As shown in Figure 3.9, we estimate the average TDC for composite applications with different service diversities under different budgets (red lines are the budget constraints).

We first analyze the algorithms in terms of their capabilities of meeting the pre-specified budget requirements. For the tightest budget, i.e., $k = 0.1$, *BHEFT* cannot generate adequate solutions for all service diversities. While as k increases, *BHEFT* is capable of deploying applications within budget, even for the composite applications with 40 different services (high diversity in Figure 3.9(c)). The three GA-based algorithms can meet all the budget levels for composite applications with all service di-



(a) The composite applications with low service diversity.

(b) The composite applications with medium service diversity.



(c) The composite applications with high service diversity.

Figure 3.9: Average TDC under different budget constraints.

versities.

Regardless of ART, we can see that the *BHEFT* approach has the lowest TDC under the majority of budget constraints ($k = 0.3, 0.4, 0.7, 0.9$ for all service diversities). *P-GA* or *CGA*^{2} has the highest TDC under nearly all budget factors and service diversities. As to our proposed *H-GA*, it shows better cost savings than *P-GA*, especially for the high budget factors, such as k =0.7 and 0.9. This is in line with the fact that *H-GA* deploys a portion of the services to the data centers with cheaper VMs described in Subsection 3.5.6.

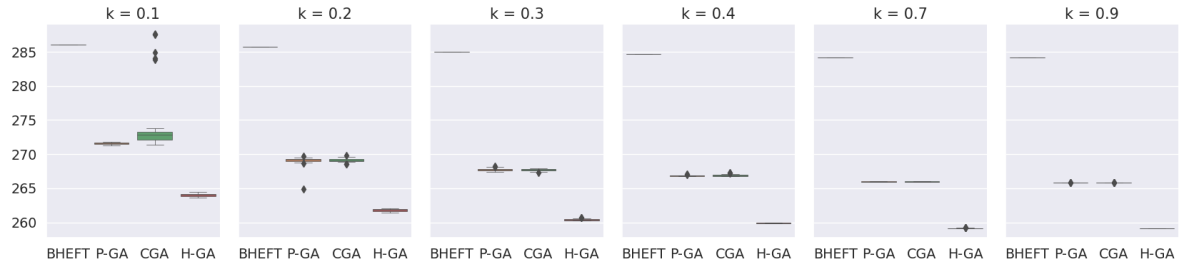
From the above results, we can conclude that *BHEFT* can obtain the lowest TDC if the budget is high, while *H-GA* is able to meet all budget requirements with relatively low TDC.

3.5.5 ART Evaluation

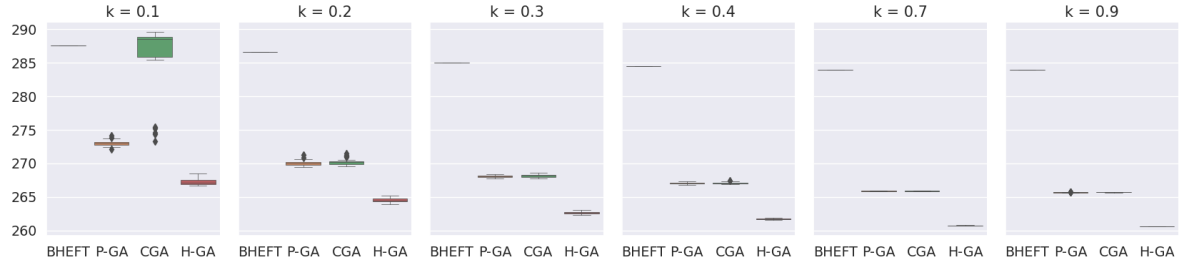
The ART generated by *BHEFT*, *P-GA*, *CGA*^{2}, and *H-GA* for composite applications with different service diversities, i.e., 20 (low), 30 (medium), and 40 (high) different services, are shown in Figure 3.10. Their mean and standard deviation of ART and TDC with different service diversities and budget factors are presented in Table 3.2.

For composite applications with low and medium service diversities, the ART achieved by the three GA-based algorithms is significantly shorter than *BHEFT* at all the budget levels. In these experiments, *H-GA* outperforms both *P-GA* and *CGA*^{2} for all cases, which shows that our proposed algorithm is more effective than the existing penalty-based methods. We also found that *CGA*^{2} has similar performance as *P-GA* in the majority of the cases. These results suggest that the adaptive penalty function in *CGA*^{2} is not effective for the *CAD* problem.

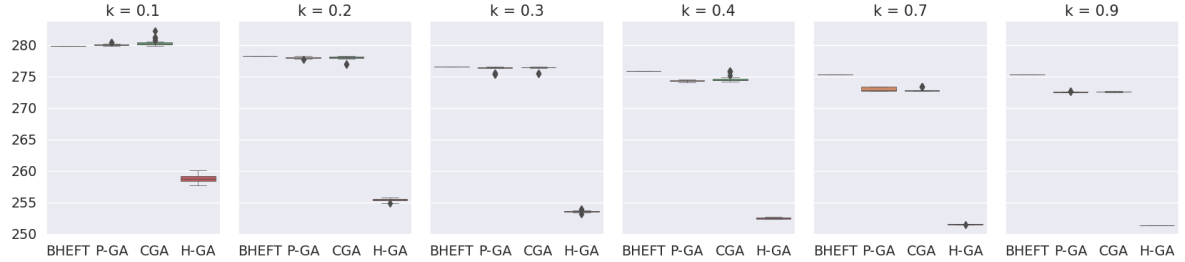
Compared with deploying composite applications with low and medium service diversities, for the high service diversity scenario, the performance improvement of *P-GA* and *CGA*^{2} than *BHEFT* is very limited, es-



(a) The composite applications with low service diversity.



(b) The composite applications with medium service diversity.



(c) The composite applications with high service diversity.

Figure 3.10: *ART* under different budget constraints.

Table 3.2: Algorithms performance comparison for deploying applications with different service diversities and budget factors (ART in ms., TDC in USD, and the best is highlighted).

Algorithm		*BHEFT*		*P-GA*		*CGA²*		*H-GA*		
Diversity	k	*ART*	*TDC*	*ART*	*TDC*	*ART*	*TDC*	*ART*	*TDC*	NoL
Low	0.1	286.01	37.38	271.58±0.14	29.15±0.12	274.73±5.02	29.26±0.17	**263.99±0.23**	29.25±0.26	2
	0.2	285.75	42.24	269.01±0.81	42.18±0.24	269.14±0.27	42±0.47	**261.76±0.2**	42.27±0.28	
	0.3	285.01	55.04	267.72±0.21	55.28±0.41	267.69±0.18	55.41±0.26	**260.37±0.12**	55.25±0.4	
	0.4	284.64	67.33	266.85±0.07	68.37±0.22	266.88±0.13	68.37±0.28	**259.89±0.04**	68.36±0.28	
	0.7	284.12	92.16	265.96±0.03	107.38±0.31	265.97±0.02	107.31±0.41	**259.16±0.02**	106.07±0.79	
	0.9	284.12	92.16	265.83±0	126.53±0.59	265.83±0	126.42±0.8	**259.14±0**	111.36±0	
Medium	0.1	287.52	41.73	273±3.87	36.95±0.37	285.9±5.27	37.01±0.23	**267.26±0.43**	36.99±0.19	3
	0.2	286.57	57.47	2160.4±3.15	57.2±0.31	270.28±0.47	57.02±0.62	**264.53±0.29**	57.23±0.28	
	0.3	284.98	77.31	2144.82±1.43	77.49±0.33	268.18±0.24	77.58±0.32	**262.66±0.17**	77.48±0.46	
	0.4	284.51	97.15	2136.48±0.95	97.85±0.42	267.08±0.14	97.77±0.36	**261.73±0.07**	97.84±0.39	
	0.7	283.96	138.24	2127.1±0.29	158.7±0.66	265.89±0.03	158.85±0.44	**260.76±0.02**	158.17±0.93	
	0.9	283.96	138.24	2125.53±0.18	186.33±3.6	265.71±0.02	186.01±3.3	**260.65±0**	174.72±0	
High	0.1	279.92	44.16	280.1±0.13	42.77±0.17	280.39±0.49	42.75±0.2	**258.81±0.55**	42.72±0.3	4
	0.2	278.35	67.58	278.03±0.13	67.33±0.33	277.98±0.3	67.36±0.26	**255.35±0.24**	67.33±0.25	
	0.3	276.54	90.88	276.32±0.35	91.87±0.23	276.41±0.25	91.77±0.3	**253.56±0.16**	91.7±0.38	
	0.4	275.91	114.69	274.37±0.12	116.36±0.46	274.69±0.51	116.14±0.65	**252.49±0.11**	116.37±0.52	
	0.7	275.34	163.07	273.03±0.3	189.71±0.92	272.9±0.25	188.83±2.33	**251.51±0.05**	177.3±0.55	
	0.9	275.32	165.89	272.55±0.04	221.68±5.75	272.59±0.05	215.79±7.44	**251.36±0**	198.86±0	

pecially when k is low. We attempt to increase the number of generations and fine-tune some parameters, without obtaining noticeably better performance. Correspondingly, *H-GA* can consistently outperform *BHEFT* by optimizing the locations of service clusters.

From Table 3.2, we can calculate the average improvement of *H-GA* in terms of ART, 8.5% lower than *BHEFT*, 2.7% lower than *P-GA*, and 2.7% lower than CGA^2 for low service diversity, 7.8% lower than *BHEFT*, 2.0% lower than *P-GA*, and 2.8% lower than CGA^2 for medium service diversity, and 8.3% lower than *BHEFT*, 7.9% lower than *P-GA*, and 8.0% lower than CGA^2 for high service diversity. The observed performance differences between *H-GA* and three comparing approaches are verified through the statistical test (Wilcoxon Rank-Sum test) with a significance level of 0.05 for all cases. We also find that *H-GA* has a smaller standard deviation than the other two GA-based algorithms, confirming that *H-GA* can con-

sistently find good deployment solutions.

From the above discussion, the following conclusions can be made from the experiments: *BHEFT* can deploy composite applications in multi-cloud quickly but not effectively, also at the risk of over-budget. Compared with *BHEFT*, GA-based algorithms are capable of generating adequate solutions and outperform *BHEFT* in terms of ART. Our proposed algorithm *H-GA* also significantly outperforms *P-GA* and *CGA*^{2}. Especially when the budget is flexible (or sufficiently large), *H-GA* can generate solutions with less TDC and shorter ART than the other two GA-based algorithms.

3.5.6 Further Analysis

We observe that all the services are deployed to the same data center in the final deployment solutions of *P-GA* and *CGA*^{2}. This is because they fail to group services by investigating interdependencies between services and application users and deploy the services in the different clusters to the different data centers. On the contrary, *H-GA* tends to deploy the services to different data centers (see the number of deployment locations, i.e., NoL in Table 3.2). This demonstrates that the service clustering mechanism plays an important role during the evolutionary search for good application deployment solutions.

Take applications with high diversity as an example, a total of 40 services are deployed to four locations, i.e., Singapore, Los Angeles, Hong Kong, and London, in the final deployment solution of *H-GA*. Specifically, some services are deployed to the cheaper VMs in Los Angeles, which can reduce TDC. On the other hand, selecting the four locations can bring service clusters close to the users, and therefore reduce the network latency of corresponding applications.

In our experiments, the performance of *H-GA* is better than *P-GA* with the service clustering mechanism, while *H-GA* significantly outperforms

CGA^2 with the service clustering mechanism. Besides, without the repair algorithm for constraint handling, *H-GA* cannot guarantee the total deployment cost within the given budget.

3.5.7 Computation Time

For the computational time, *BHEFT* can generate solutions quickly (within 0.1 seconds for all the test scenarios). The average computation time of *H-GA* with respect to all service diversities ranges from 75 to 200 seconds, which is slightly less than *P-GA*. Because *CGA*^{2} involves the co-evolution of crossover and mutation rates, its average computation time is about 10 times longer than *H-GA* and *P-GA* (the overall evolution complexity is $O(M_1G_1M_2G_2)$).

To verify the effectiveness of the dynamic upper bound mechanism, we implement the algorithm that repairs all over-budget solutions. The algorithm cannot achieve better performance in terms of ART, but spends on average 29.66% more average computation time (ACT shown in Table 3.3). The results illustrate that the mechanism of dynamic repair upper bound can trade off performance with computation time well.

Besides, for the highest diversity with the tightest budget, *H-GA* can generate the solution within 13 minutes. That is, if the request rate distribution does not change significantly within an hour (as confirmed by many existing studies [23], [135]), periodical use of *H-GA* is highly feasible in practice, even considering the extra time required for application migration (previous research shows that service migrations in the cloud can usually be performed within 15 minutes [106]). Note that a shorter service migration time can further improve the effectiveness of *H-GA*.

Table 3.3: Algorithms performance comparison with different repair mechanisms (ART in ms. and ACT in s.).

		Adaptive bound		Fully repair		Δ	
Diversity	k	ART	ACT	ART	ACT	ART	ACT
Low	0.1	263.99	247.98	263.96	351.57	0.01%	-29.46%
	0.2	261.76	106.11	261.76	197.09	0.00%	-46.16%
	0.3	260.37	44.43	260.42	98.84	-0.02%	-55.05%
	0.4	259.89	24.60	259.88	49.75	0.00%	-50.55%
	0.7	259.16	21.73	259.16	22.85	0.00%	-4.90%
	0.9	259.14	21.76	259.14	22.87	0.00%	-4.85%
Medium	0.1	267.26	596.21	267.48	713.6	-0.08%	-16.45%
	0.2	264.53	250.41	264.44	394.78	0.04%	-36.57%
	0.3	262.66	77.69	262.80	175.32	-0.05%	-55.69%
	0.4	261.73	28.98	261.69	65.67	0.01%	-55.87%
	0.7	260.76	24.80	260.78	26.04	-0.01%	-4.76%
	0.9	260.65	24.82	260.65	25.99	0.00%	-4.50%
High	0.1	258.81	765.05	258.72	884.65	0.03%	-13.52%
	0.2	255.35	281.10	255.44	454.53	-0.03%	-38.16%
	0.3	253.56	66.72	253.51	172.72	0.02%	-61.37%
	0.4	252.49	26.37	252.51	49.04	-0.01%	-46.23%
	0.7	251.51	25.71	251.51	27.33	0.00%	-5.93%
	0.9	251.36	25.69	251.36	26.72	0.00%	-3.85%

3.6 Chapter Summary

In this chapter, we study the *CAD* problem in multi-cloud that minimizes the average response time subject to a budget constraint. To address the problem, we propose a novel GA-based algorithm that can select VMs in multi-cloud for constituent services of composite applications within a given cost budget so that the average response time of all composite applications is minimized. Due to the immense search space resulting from the diversity of VM locations and types, we design the service clustering algorithm to reduce the search space by effectively utilizing the information hidden in application structures. To improve solution quality and con-

trol the computation time of the algorithm, we develop a problem-specific repair algorithm to transform a self-adaptive subset of over-budget solutions into budget feasible solutions. Furthermore, our proposed algorithm *H-GA* applies the two-string decoding and domain-tailored population initialization, which are effective for breeding excellent offspring. The experiments based on the datasets collected from real-world cloud providers, network environments, and business applications show that our algorithm can meet the budget and achieve up to about 8% performance improvement in terms of ART compared with existing algorithms.

We find that the performance improvement through increasing budgets is limited (shown in Table 3.2). Taking the applications with the high diversity as an example, *H-GA* only can reduce ART from about 259 ms to 251 ms when the budget is increased from about 43 USD to 199 USD. Next chapter will study *ARD* in multi-cloud. Different from *CAD*, *ARD* applies service-oriented replication to further reduce ART below a stringent threshold, e.g., 150ms. This target ART will guarantee the satisfactory Quality of Experience (QoE) of cloud applications [201].

Chapter 4

Application Replication and Deployment

4.1 Introduction

This chapter tackles Application Replication and Deployment (*ARD*) in multi-cloud. For some applications, a low average response time must be satisfied to guarantee the quality of experience (QoE). Therefore, application providers must replicate applications in multiple locations to ensure the stringent requirement on average response time. While ensuring the acceptable performance of applications, application providers usually care about cost minimization [104], [201]. Therefore, we consider the *ARD* problem as a constrained optimization problem to minimize the total deployment cost subject to the constraints on the average response time.

Different from *CAD*, the deployment objects of *ARD* are application replicas rather than constituent services. Particularly, the *ARD* problem has several different concerns. On the one hand, we need to determine the number of replicas for each application before selecting proper VMs in multi-cloud for application deployment. On the other hand, we need to determine how to dispatch widely distributed user requests among different application replicas.

In this chapter, we consider two different approaches for request dispatching. One of the two approaches is to dispatch user requests to the closest application replicas, which is a common practice in the literature [74], [171] and industry [113]. To further reduce the deployment cost, we apply another request dispatching approach that allows requests from the same user region to be served by different application replicas [201]. We represent the two request dispatching approaches as *close dispatching* and *flexible dispatching* respectively.

According to *close dispatching*, the location of application replicas, i.e., *replica placement*, determines the dispatching of user requests and further the average response time of applications. Besides, the replica placement significantly impacts the deployment cost of applications due to the differentiated price of VMs in different data centers. We design a new two-level optimization approach under the GA framework to solve the *ARD with close dispatching* problem. The main algorithm novelties are summarized as follows.

- Considering various data center locations and applications results in extremely large search spaces. Therefore, the first level optimization, i.e., replica placement in multi-cloud, is realized by GA.

- To reduce the complexity/size of the search space, the second level optimization, i.e., *VM types selection* for application deployment, is performed through a domain-tailored heuristic. Particularly, after deploying all application replicas to the cheapest VMs that satisfy the capacity requirement, we progressively upgrade the VM types with the hope to reduce the response time as long as the performance requirement is satisfied.

- We initialize the population of the GA-based approach by adding a heuristic-based solution to improve solution quality and convergence speed.

According to *flexible dispatching*, we need to simultaneously optimize the replica deployment and request dispatching. We design a two-stage optimization process to solve the *ARD with flexible dispatching* problem. The new algorithm design process is summarized as follows.

- We first transfer the problem to a series of MILP problems so that it can be solved by off-the-shelf MILP methods.
- Next, we propose a MILP-based algorithm to efficiently obtain a high-quality base solution. Following a novel double iterative mechanism, the algorithm adaptively updates the upper bounds on both the VMs' utilization rate and average response time to improve the performance of the base solution. The worst-case performance of the base solution will also be theoretically analysed.
- To further reduce the total deployment cost, we develop an LNS-based algorithm to improve the base solution. To build an effective LNS process, a new destroy heuristic is proposed to remove a portion of application replicas. Then a problem-specific repair heuristic is designed to effectively deploy new application replicas. Besides, we propose a delay-oriented heuristic to dispatch user requests to achieve high performance based on currently deployed replicas.

The main contributions of this chapter are summarized as follows:

Firstly, we formally define the *ARD* problem in multi-cloud as a constrained optimization problem with the goal to minimize the total deployment cost subject to constraints on average response time. To the best of our knowledge, this is the first study in the literature on the application deployment problem with the consideration of location-aware replication for deploying interactive applications in multi-cloud.

Secondly, we propose a GA-based approach, namely *GA-ARD*, to solve the *ARD with close dispatching* problem. *GA-ARD* features a newly designed and domain-tailored solution representation, fitness measurement,

and population initialization. Extensive experiments based on the real world datasets show *GA-ARD* can shorten the average response time of *H-GA* and significantly reduce deployment cost compared with the common application replication and placement strategies in [88].

Finally, we propose an optimization approach, i.e., *MCApp*, to solve the *ARD with flexible dispatching* problem. With the extra optimization decision, i.e., the request dispatching, the problem is firstly linearized by bounding the utilization rates of the deployed VMs. Then, *MCApp* creates a hybrid optimization process that combines an iterative MILP-based algorithm and a domain-tailored LNS-based algorithm to simultaneously optimize replica deployment and request dispatching. *MCApp* is compared to several state-of-the-art approaches, including *LNS-MC* [68] and *HMOHM* [109]. The experimental results indicate that *MCApp* significantly outperforms the existing approaches, achieving up to 25% savings in terms of the total deployment cost.

4.2 Chapter Organization

The remainder of this chapter is organized as follows. Section 4.3 presents the *ARD* problem formulation. Section 4.4 introduces the *GA-ARD* approach for the *ARD with close dispatching* problem. The chromosome representation, fitness measurement, population initialization, and genetic operators are all described in this section. To evaluate *GA-ARD*, we conduct a series of experiments in Section 4.5. Section 4.6 introduces the *MCApp* approach for the *ARD with flexible dispatching* problem. The problem linearization, MILP-based algorithm, and LNS-based algorithm are detailed in this section. In Section 4.7, we conduct extensive experiments to evaluate *MCApp*. Section 4.8 summarizes this chapter.

4.3 *ARD* Problem Formulation

The aim of the *ARD* problem is to select VMs from multi-cloud for application replication and deployment to minimize the Total Deployment Cost (TDC) subject to a stringent performance requirement on Average Response Time (ART). The key notations to be used for problem definition are listed in Table 4.1.

An application provider delivers a suite of applications $\mathcal{A}$ for its users distributed in global user regions $\mathcal{U}$. During an epoch, e.g., an hour [23], the request rate γ_{au} denotes the request frequency for application A_a ($A_a \in \mathcal{A}$) from user region U_u ($U_u \in \mathcal{U}$).

We consider a set of VM types $\mathcal{V}$ and a set of multi-cloud data center $\mathcal{C}$. If VM type V_v ($V_v \in \mathcal{V}$) is available in data center C_c ($C_c \in \mathcal{C}$), we use rc_{vc} to denote its rental cost per time unit and δ_{av} to measure the maximum amount of requests for application A_a processable by V_v per time unit.

Referring to [139], [212], we assume that all requests for application A_a dispatched to data center C_c will be served by a single replica of A_a in C_c. In the remaining of this chapter, we use R_{ac} to refer to the application replica.

Let $x_{auc} \in [0, 1]$ denote the percentage of requests from user region U_u for A_a that will be dispatched to data center C_c. The *workload* of R_{ac} can be measured by combining relevant request rates from all user regions:

$$\lambda_{ac} = \sum_{u=0}^{|\mathcal{U}|-1} \gamma_{au} x_{auc}. \tag{4.1}$$

Referring to [109], we assume that each application replica is deployed to one instance of one VM type V_v in multi-cloud. For the application replica with multiple constituent services, we deploy them as microservices [193] in the same VM instance to make the inter-services network latency negligible. We further assume that application replicas apply the eventual consistency model when they access the database since the consistency model is commonly used by widespread cloud applications with

Table 4.1: Mathematical notations for the *ARD* problem

Notation	Definition
A_a	The a^{th} application
U_u	The u^{th} user center
V_v	The v^{th} type of VM
C_c	The c^{th} data center
R_{ac}	Replica of application A_a in data center C_c
γ_{au}	The request rate of A_a from the u^{th} user center
δ_{av}	Amount of requests for application A_a processable by VM type V_v per time unit
rc_{vc}	The rental cost of VM type V_v in data center C_c
dt_{uc}	Round-trip delay between user region U_u and data center C_c
l	Overall request rate across all applications
m	Maximum average response time per request
x_{auc}	Independent variables: percentage of application A_a request rate from user region u to be served in data center C_c
y_{acv}	Independent variables: to rent an instance of VM type V_v for application replica R_{ac} (1) or not (0)
λ_{ac}	Dependent variables: workload of application replica R_{ac}
μ_{ac}	Dependent variables: capacity of application replica R_{ac}
pt_{ac}	Dependent variables: average request processing time of application replica R_{ac}
z_{ac}	Independent variables: to deploy a replica of application A_a in data center (1) or not (0)
n	Dependent variable: total replica number for all applications
u_{ac}	Dependent variables: utilization rate of the VM when deploying application replica R_{ac}

large numbers of user requests [121], [189]. Let binary variable y_{acv} indicate whether an instance of V_v is selected to deploy application replica R_{ac}, the *capacity* of R_{ac} depends on the processing speed of the selected VM type by:

$$\mu_{ac} = \sum_{v=0}^{|\mathcal{V}|-1} \delta_{av} y_{acv}, \tag{4.2}$$

where $\sum_{v=0}^{|\mathcal{V}|-1} y_{acv} \leqslant 1$ $(A_a \in \mathcal{A}, C_c \in \mathcal{C})$. There are two situations for eq. (4.2). If there is a replica of A_a deployed in data center C_c, $\mu_{ac} = \delta_{av}$ where $y_{acv} = 1$. Otherwise, $\mu_{ac} = 0$.

We follow [187] and model the operation of each individual application replica as an $M/M/1$ queue. According to Little's Law [98], the average request processing time of replica R_{ac} depends on both μ_{ac} and λ_{ac}:

$$pt_{ac} = \frac{1}{\mu_{ac} - \lambda_{ac}}. \tag{4.3}$$

Let dt_{uc} represent Round-Trip Delay (RTD) between user region U_u and data center C_c, we can calculate the average response time of application set $\mathcal{A}$ by:

$$ART = \frac{\sum_{a=0}^{|\mathcal{A}|-1} \sum_{c=0}^{|\mathcal{C}|-1} \sum_{u=0}^{|\mathcal{U}|-1} \gamma_{au} x_{auc} (dt_{uc} + pt_{ac})}{l}, \tag{4.4}$$

where $l = \sum_{a=0}^{|\mathcal{A}|-1} \sum_{u=0}^{|\mathcal{U}|-1} \gamma_{au}$ is the total request rate across the application set $\mathcal{A}$.

With the goal to minimize TDC of $\mathcal{A}$ in multi-cloud, we have two decision variable vectors, i.e. *request dispatch plan* X and *replica deployment plan* Y. Here, X determines how users' application requests are dispatched among all data centers, i.e., x_{auc}, and Y determines the types of VMs and the corresponding data centers to deploy all application replicas, i.e., y_{acv}. Concretely, the *ARD* problem is formulated as follows:

Problem 1.

$$\min \quad TDC = \sum_{a=0}^{|\mathcal{A}|-1} \sum_{c=0}^{|\mathcal{C}|-1} \sum_{v=0}^{|\mathcal{V}|-1} y_{acv} rc_{vc} \tag{4.5}$$

subject to:

(a) $y_{acv} \in \{0, 1\} \quad A_a \in \mathcal{A}, C_c \in \mathcal{C}, V_v \in \mathcal{V}$

(b) $\sum_{v=0}^{|\mathcal{V}|-1} y_{acv} \leqslant 1 \quad A_a \in \mathcal{A}, C_c \in \mathcal{C}$

(c) $ART \leqslant m$

(d) $pt_{ac} > 0 \quad A_a \in \mathcal{A}, C_c \in \mathcal{C}$

(e) $0 \leqslant x_{auc} \leqslant 1 \quad A_a \in \mathcal{A}, U_u \in \mathcal{U}, C_c \in \mathcal{C}$

(f) $\sum_{c=0}^{|\mathcal{C}|-1} x_{auc} = 1 \quad A_a \in \mathcal{A}, U_u \in \mathcal{U}$

Constraint (a) indicates whether an instance of VM type V_v in data center C_c is rented for hosting application A_a. Constraint (b) guarantees that each application replica is deployed to one VM instance. Constraint (c) guarantees that ART of the application set $\mathcal{A}$ is below the acceptable threshold m set by the application provider. Constraint (d) guarantees that $\mu_{ac} > \lambda_{ac}$, i.e., the capacity of any application replica must be sufficient to process its workload. Constraints (e) and (f) guarantee that every request for application A_a from user region U_u will be processed. Note that **Problem 1** is nonlinear due to constraint (c).

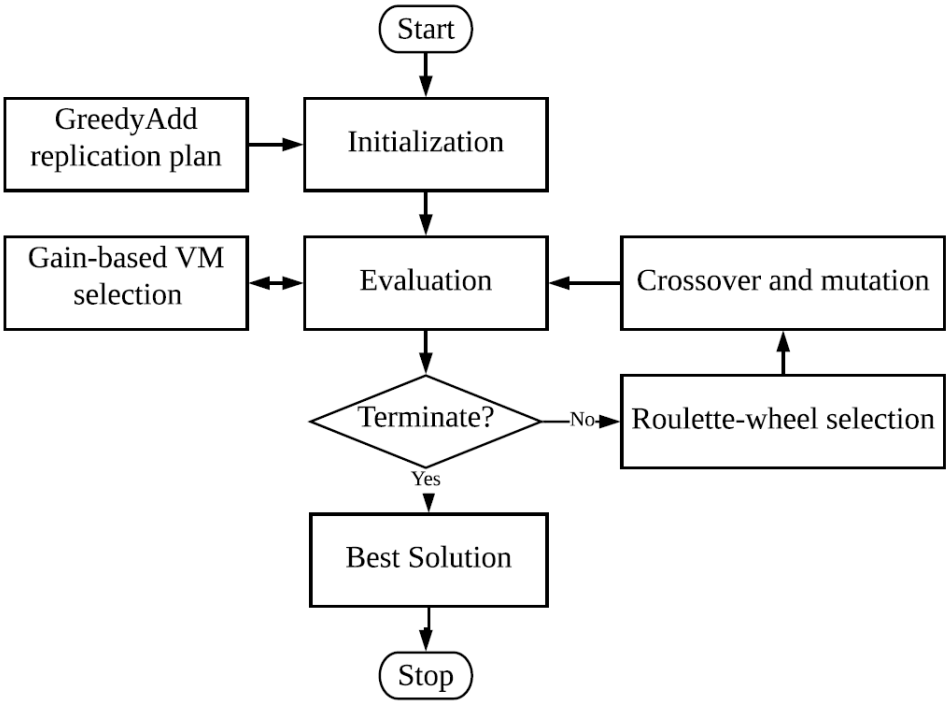


Figure 4.1: The overall process of GA-ARD.

4.4 *GA-ARD* for the *ARD with Close Dispatching* Problem

According to close dispatching, the request dispatching depends solely on the locations of application replicas, i.e., the replica placement. Let binary variable z_{ac} indicate whether application A_a is deployed in data center C_c, we have:

$$x_{auc} = \begin{cases} 1 & \text{if } dt_{uc} = \underset{i:z_{ai}=1}{\text{argmin}}\{dt_{ui}\} \\ 0 & \text{otherwise.} \end{cases} \tag{4.6}$$

In this section, we propose our approach under the GA framework, named *GA-ARD*, to solve the *ARD with close dispatching* problem. The overall process of *GA-ARD* is shown in Figure 4.1. In *GA-ARD*, the op-

timization of *replica placement plans*, i.e., $Z = [z_{ac}]$, is realized by GA. After encoding replica placement plans, a heuristic-based plan is included in the randomly generated initial population (described in Subsection 4.4.3). For each encoded plan, *GA-ARD* optimizes VM type selection of all application replicas and calculates ART of the deployment solution as the fitness value (described in Subsection 4.4.2). After the roulette-wheel selection, the crossover and mutation operations are performed to generate new plans. The population is evolved iteratively until the predefined maximum number of generations is reached. Finally, the best chromosome is decoded to return the final replication and deployment decision. In the following subsections, we provide a detailed description of *GA-ARD*, including representation, fitness measurement, population initialization, and genetic operators.

4.4.1 Chromosome Representation

We use chromosomes to encode replica placement plans evolved by GAs. Here, we transform the replica placement plan, i.e., the matrix Z into a bit string as shown in Figure 4.2. The length of the string is $|\mathcal{A}| \cdot |\mathcal{C}|$. For the example chromosome in Figure 4.2, there are two applications and ten data centers. The three replicas of application A_0 are deployed at C_0, C_2, and C_8, and the two replicas of application A_1 are deployed at C_1 and C_5.

4.4.2 Fitness Measurement

For the randomly generated and newly evolved chromosomes that represent replica placement plans, *GA-ARD* applies a heuristic to optimize VM type selection with the minimal TDC while ensuring that ART is below m. For each chromosome, we regard the TDC as its fitness. The fitness value of a chromosome indicates its chance of survival and reproduction in the next generation.

The heuristic to optimize VM type selection is inspired by the GAIN

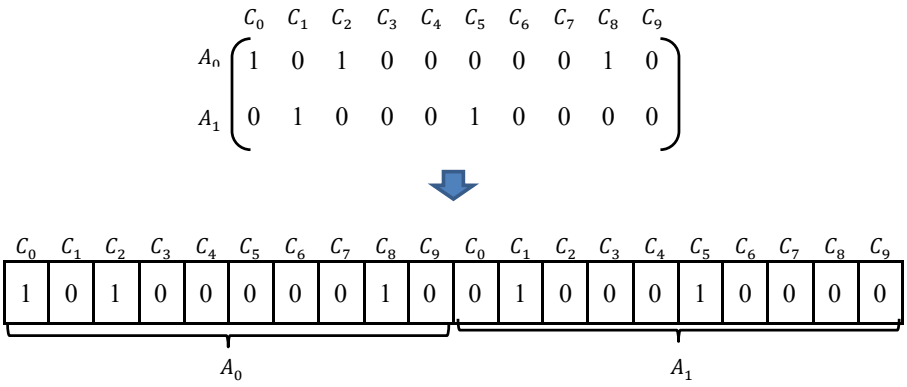


Figure 4.2: An example chromosome

approach in [145], which is a heuristic-based scheduling approach considering constraints. The proposed heuristic aims to efficiently generate a constraint-compliant VM type selection with low TDC by the following steps:

- Select the cheapest capacity feasible VM types for all application replicas;
- Calculate the *benefit* for each application replica by upgrading its current VM type to one with higher capacity, e.g., from AWS m5.large to m5.xlarge. The *benefit* is calculated as follows:

$$benefit = \frac{ART_{old} - ART_{new}}{TDC_{new} - TDC_{old}}, \tag{4.7}$$

  where ART_{new} and TDC_{new} are the response time and deployment cost after upgrade;
- Upgrade the application replica with the largest benefit iteratively until the ART is within the given m. If ART cannot be brought within m even if all replicas are deployed to the VM type with the highest capacity, we assign the fitness of the chromosome as ∞ (the smaller the better).

4.4.3 Population Initialization

To improve the solution quality and convergence speed, we propose a heuristic-based method, named *GreedyAdd*, to generate a solution to be included in the initial population as discussed below. The remaining population will be initialized randomly. That is, we randomly select some data centers as the locations of application replicas.

Refer to the incremental replication method in [94], *GreedyAdd* aims to progressively identify an appropriate number of application replicas as follows:

- For each application A_a, we deploy its first replica to the data center with the minimal average RTD:

$$ARTD_a = \frac{\sum_{c=0}^{|\mathcal{C}|-1} \sum_{u=0}^{|\mathcal{U}|-1} \gamma_{au} x_{auc} dt_{uc}}{\sum_{u=0}^{|\mathcal{U}|-1} \gamma_{au}}. \tag{4.8}$$

- While ART of the current solution is within m, one additional replica is progressively allocated to one of the remaining data centers, where ART can achieve the highest reduction.

- To further improve the solution in terms of TDC, we attempt to progressively allocate more replicas to the remaining data centers, where TDC can be reduced while ensuring ART is under m. The process is repeated until we cannot find better solutions.

Note that the VM type selection for evaluating ART applies the Gain-based heuristic as described in Subsection 4.4.2.

4.4.4 Crossover Operator

We apply the crossover operator to evolve replica placement plans. During crossover operation, the parent chromosomes are divided into two

parts by one random cut-off point. Their offsprings are generated by exchanging tail parts of the two parents as the example in Figure 4.3. Note that to avoid generating an invalid plan, i.e., the replica number of any application after the crossover operation is 0, the crossover is only performed on bit locations with 1s in the parent chromosomes. Accordingly, those bit locations with 0s can be easily filled up after crossover.

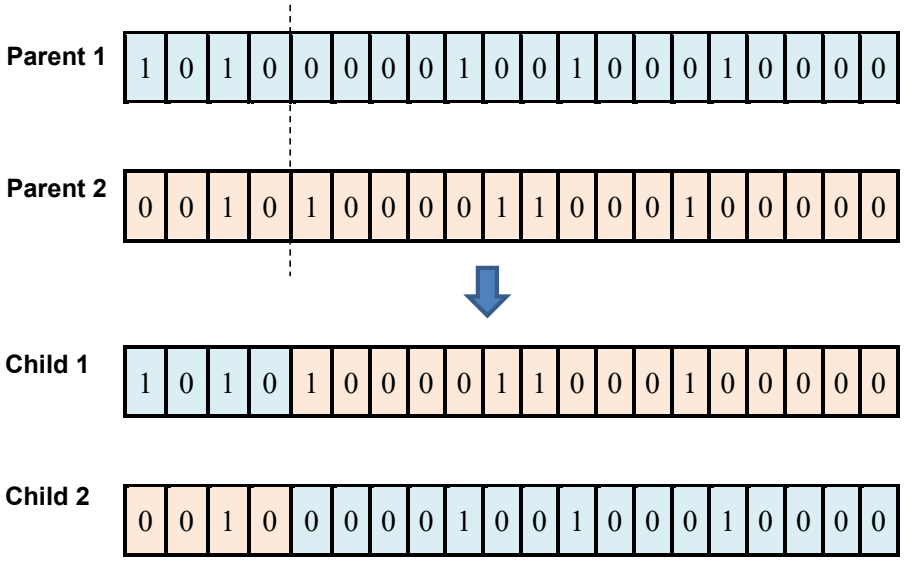


Figure 4.3: An example of crossover operation

4.4.5 Mutation Operator

The mutation operator can introduce a local improvement to previously evolved plans by making a small change on an offspring in a random way. In particular, a bit is selected randomly and inverted, i.e., if the bit is 1, it is changed to 0 and vice versa. Similarly, the operation may generate an invalid plan without replicas for any A_a. In this case, we invert a new random bit until a valid plan is generated.

4.5 Evaluation of *GA-ARD*

We conduct a series of experiments to evaluate the performance of *GA-ARD* by comparing it with the common placement and replication strategies [88], e.g., deploying the replicas based on the distance between data centers and users. Also, we adapt our proposed *H-GA* in Section 3.4 to the *ARD* problem to evaluate the effectiveness of application replication. The datasets used in the section are the same as the datasets introduced in Subsection 3.5.1.

4.5.1 Experiment Settings

In our experiments, 10 application sets are generated by randomly selecting enterprise applications based on [75]. The range of application processing time for a single request is 10-20 ms running on AWS m5.large VM. The number of applications in an experimented set ranges from 1 to 10 to simulate a wide variety of user requirements. These application numbers are sufficient for enterprise applications, such as Mercedes-Benz.io[1] and MetService[2]. Particularly, MetService, the meteorological service of New Zealand, provides 8 applications for different user groups, i.e., National Forecast, Towns and Cities Forecast, Rural Forecast, Marine Forecast, Mountains and Parks Forecast, Maps and Radar Information, Warnings Information, and Public Information. Referring to [201], the acceptable threshold of ART, i.e., m, is set to 150 ms.

4.5.2 Competing Approaches and Parameter Settings

The distance-based replica placement and replication strategy in [88] include three settings. For each application, (1) one replica in the closest data center from users (Distance 1), (2) two replicas in the first and second

[1]https://www.mercedes-benz.io/
[2]https://www.metservice.com/

closest data centers (Distance 2), and (3) three replicas in the three closest data centers (Distance 3). This strategy shows the trade-off between the total deployment cost and the average response time in terms of the number of replicas. That is, more replicas can reduce the average response time but increase the total deployment cost. After checking the three settings, we use *Distance 3* as the baseline since this setting performs best in terms of ART satisfaction and TDC minimization. The *Full* placement and replication strategy in [88], i.e., deploying replicas at all data centers, is also included to benchmark the performance of *GA-ARD*.

To apply *H-GA* to the *ARD* problem, we change its time-based fitness function to TDC and budget-related constraint to ART. For both *GA-ARD* and *H-GA*, our parameter settings include: population size is 100, maximum generation is 100, which are sufficient for the search process to converge. The crossover rate and mutation rate are 0.9 and 0.1 respectively following common practice in the literature [103].

To compare the results, we consider the mean and standard deviation of TDC after running each experiment 30 times on the same computer with Intel Core i7-8700 CPU (3.2 GHz and 16 GB of RAM).

4.5.3 Performance Comparison

Because *Distance 3*, *Full*, and *GA-ARD* can all satisfy the constraints on ART, we show the average TDC achieved for ten problem instances in Figure 4.4. The mean and standard deviation of TDC are presented in Table 4.2.

From the experimental results, we observe that *Distance 3* outperforms *Full* when the number of applications $|\mathcal{A}| > 1$. These results suggest that except for single application deployment, it is not effective to replicate applications in all data centers. In such a situation, many application replicas only serve a small number of users, which causes the leased VMs underutilized.

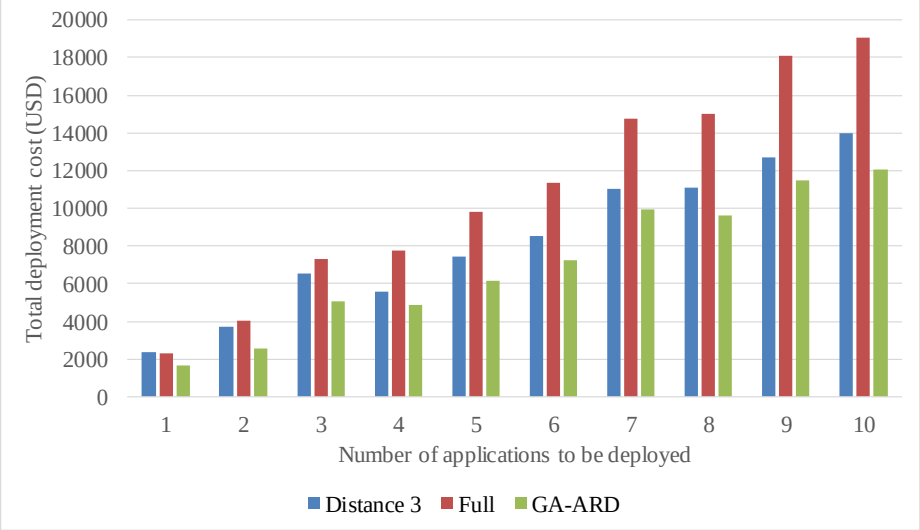


Figure 4.4: Comparison of average TDC by competing approaches.

GA-ARD can generate the best solutions among competing approaches with respect to all problem instances. From Table 4.2, we can calculate that the average improvement of *GA-ARD* in terms of TDC is 17.5%, in comparison to *Distance 3*, e.g., 30.62% when $|\mathcal{A}| = 2$ (highest) and 9.76% when $|\mathcal{A}| = 7$ (lowest), and 34.64% lower than *Full*, e.g., 37.23% when $|\mathcal{A}| = 5$ (highest) and 27.58% when $|\mathcal{A}| = 1$ (lowest). The observed performance differences between *GA-ARD* and two competing approaches are verified by the statistical test (Wilcoxon Rank-Sum test) with the significance level of 0.05 for all problem instances. We also find that *GA-ARD* has very small standard deviations, confirming its stability and reliability for the *ARD with close dispatching* problem.

4.5.4 Effectiveness of Application Replication

In this subsection, we evaluate the effectiveness of application replication for satisfying constraints on ART. We show the ART achieved by *GA-ARD*

Table 4.2: Performance comparison of the approaches for the *ARD with close dispatching* problem with different application diversities (TDC per month in USD, the best is highlighted).

No. of Apps	*Distance 3* [88]		*Full* [88]		*GA-ARD*	
	TDC	n^*	TDC	n^*	TDC	n^*
1	2376±0	3	2294.64±0	15	**1661.76±0**	4 ↓
2	3732.48±0	6	4024.8±0	30	**2589.6±21.62**	7 ↓
3	6549.12±0	9	7295.76±0	45	**5049.12±121.38**	17 ↑
4	5597.28±0	12	7768.8±0	60	**4881.6±24.94**	13 ↑
5	7416±0	15	9781.92±0	75	**6140.16±0**	17
6	8557.92±0	18	11370.24±0	90	**7227.36±0**	21
7	11007.36±0	21	14735.52±0	105	**9933.12±0**	24
8	11067.84±0	24	15036.48±0	120	**9644.4±19.05**	29 ↑
9	12718.08±0	27	18084.24±0	135	**11472.48±0**	29
10	13957.92±0	30	19037.52±0	150	**12034.08±6.6**	34 ↑

and *H-GA* in Figure 4.5. For all the experiments with different numbers of applications, *H-GA* cannot generate solutions having ART within 150 ms, i.e., the red dotted line in Figure 4.5. For example, the minimal ART achieved by *H-GA* is 238.18 ms for the problem instance with $|A| = 5$ and the maximal ART achieved by *H-GA* is 244.93 ms for the problem instance with $|A| = 9$. *GA-ARD* is capable of replicating and deploying applications with ART strictly below 150 ms.

If most users of an application are geographically localized, *H-GA* may generate adequate solutions without replication. When serving the requests from widely distributed users, *H-GA* cannot bring ART under 150 ms even selecting the most expensive VMs for all applications. In that case, deploying application replicas at different data centers is imperative to reduce response time [135]. Therefore, we study the *ARD* problem in this thesis.

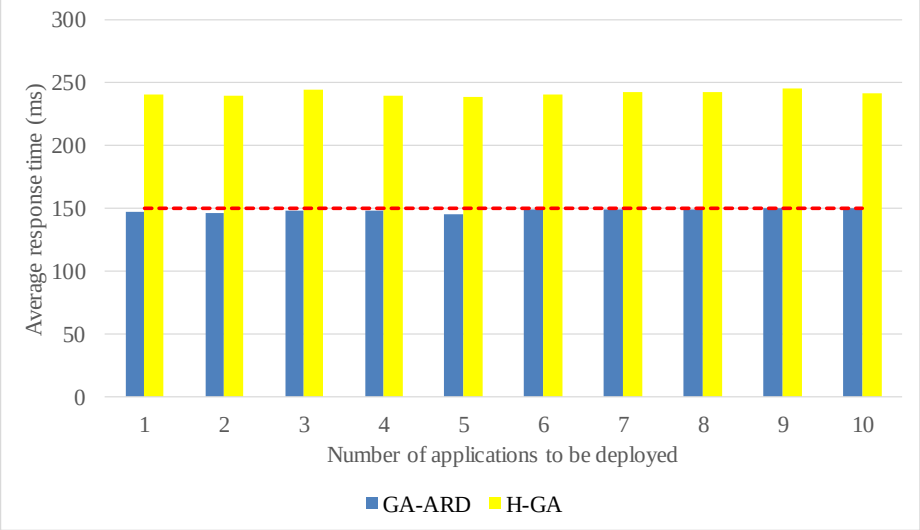


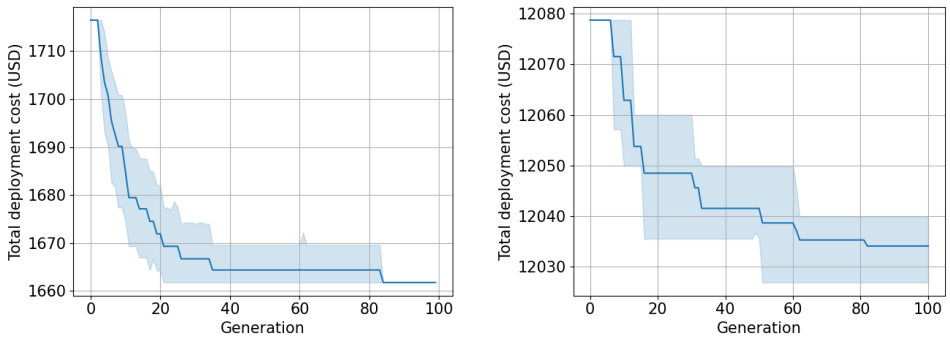
Figure 4.5: Comparison of ART for evaluating the effectiveness of application replication.

4.5.5 Replica Amount

Next, we investigate the impact of replica numbers on the solution performance. The replica number n^* of the best *GA-ARD* solution among 30 runs is presented in Table 4.2.

In most experiments, *GA-ARD* deploys slightly more application replicas than *Distance 3*. For example, when $|\mathcal{A}| = 1$, *Distance 3* identifies Northern Virginia, Singapore, and Sao Paulo as the three closest data centers to users. Accordingly, *GA-ARD* selects Northern Virginia, Seoul, Tokyo, and Mumbai to deploy application replicas. The different replica numbers and the locations are significant to reduce TDC.

We further compare n^* with the replica numbers obtained by *GreedyAdd*, i.e., the heuristic-based method introduced in Subsection 4.4.3. There are 6 cases where n^* is different. Concretely, *GA-ARD* selects more replicas in 4 cases shown as $\uparrow$ in Table 4.2, and fewer replicas in 2 cases shown as



(a) 1 application to be deployed (b) 10 applications to be deployed

Figure 4.6: Changes of TDC with *GA-ARD*

$\downarrow$. By observing the replica placement plan generated by *GreedyAdd* when $|\mathcal{A}| = 1$, the five application replicas are progressively placed in Singapore, Northern Virginia, Tokyo, Mumbai, and Seoul. However, *GA-ARD* can escape from this local optimal point achieved by the heuristic-based method. Particularly, *GA-ARD* removes the application replica in Singapore. The change saves more cost of application deployment.

4.5.6 Further Analysis

The average computation time of *GA-ARD* with respect to different budget levels is within 1500 seconds. Therefore, *GA-ARD* is applicable to the scenarios where applications have stable or predictable demand [135], e.g., the workload of applications does not change significantly within an hour or can be predicted one hour ahead.

We plot the evolution of TDC corresponding to the cases where 1 and 10 applications will be deployed (Figure 4.6) by *GA-ARD*. The same convergence behavior has been witnessed in other cases. As evidenced in the two figures, *GA-ARD* can converge within 100 generations.

4.6 *MCApp* for the *ARD with Flexible Dispatching* Problem

The *ARD with close dispatching* problem can be considered as a special case of the *ARD with flexible dispatching* problem. To support flexible dispatching, we must jointly optimize both the *request dispatch plan* X and the *replica deployment plan* Y. In eq. (4.3), the average request process time pt_{ac} is determined by the workload μ_{ac} and capacity λ_{ac} for application replica R_{ac}. Note that μ_{ac} and λ_{ac} together determine the utilization rate of the deployed VM instance for R_{ac}. We can linearize **Problem 1** by bounding utilization rates of the VMs to obtain a high-quality base solution to **Problem 1**. Inevitably, such the upper bound reduces the solution space for **Problem 1**. Hence, the base solution is not necessarily the optimal solution to **Problem 1**. Taking advantage of both exploration and propagation, LNS has recently shown outstanding performance in solving various scheduling problems due to its flexibility in designing problem-specific destroy and repair heuristics [132]. In view of this, we further design a problem-specific LNS algorithm to improve the base solution.

In this section, we introduce our *MCApp* approach (see Figure 4.7). It first transforms **Problem 1** to a series of MILP problems in Subsection 4.6.1. These linearized problems are iteratively solved through a MILP-based algorithm to obtain the base solution in Subsection 4.6.2. Finally, an LNS-based algorithm to improve the base solution is introduced in Subsection 4.6.3.

4.6.1 Problem Transfer

MCApp linearizes **Problem 1** making it solvable by off-the-shelf MILP methods. We first determine the utilization rate of the VM instance for application replica R_{ac} by:

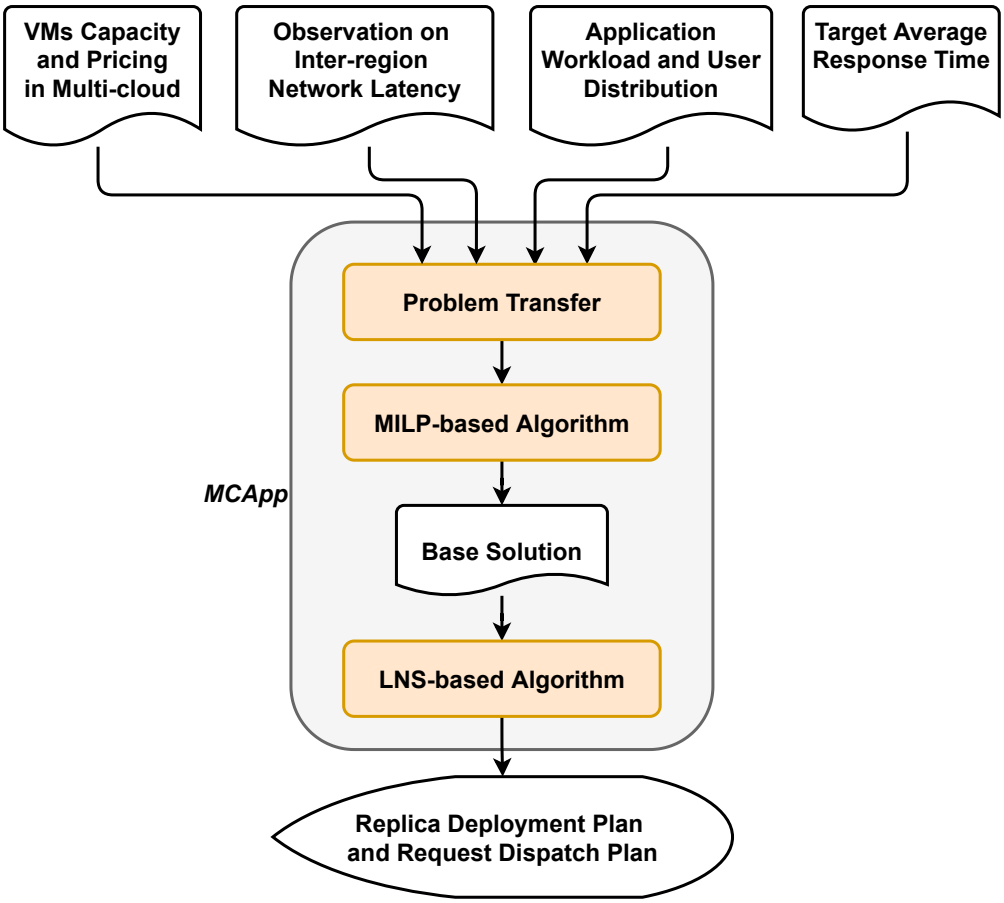


Figure 4.7: Overview of *MCApp* with inputs and outputs.

$$u_{ac} = \frac{\lambda_{ac}}{\mu_{ac}}, \tag{4.9}$$

where $0 \leqslant u_{ac} < 1$ based on constraint (a), (d), and (e) in **Problem 1**. According to eq. (4.9), the average request processing time of R_{ac} in eq. (4.3) can be rewritten as:

$$pt_{ac} = \frac{1}{\mu_{ac} - \lambda_{ac}} = \frac{1}{\frac{\lambda_{ac}}{u_{ac}} - \lambda_{ac}} = \frac{1}{\lambda_{ac}(\frac{1}{u_{ac}} - 1)} = \frac{1}{\lambda_{ac}} \cdot \frac{u_{ac}}{1 - u_{ac}}.$$

Therefore, in eq. (4.4) we have:

$$\sum_{u=0}^{|\mathcal{U}|-1} \gamma_{au} x_{auc} pt_{ac} = \lambda_{ac} pt_{ac} = \frac{u_{ac}}{1-u_{ac}}.$$

In the above, $f(u_{ac}) = \frac{u_{ac}}{1-u_{ac}}$, is monotonically increasing with respect to $u_{ac} \in [0, 1)$. Let the constant $\hat{u}_{ac}$ denote the upper bound on u_{ac} and $\hat{u} = \max_{A_a \in \mathcal{A}, C_c \in \mathcal{C}} \{\hat{u}_{ac}\}$, we have:

$$\sum_{u=0}^{|\mathcal{U}|-1} \gamma_{au} x_{auc} pt_{ac} = \frac{u_{ac}}{1-u_{ac}} \leqslant \frac{\hat{u}_{ac}}{1-\hat{u}_{ac}} \leqslant \frac{\hat{u}}{1-\hat{u}}. \tag{4.10}$$

Therefore, **Problem 1** can be transferred as follows:

Problem 2.

$$\min \quad TDC = \sum_{a=0}^{|\mathcal{A}|-1} \sum_{c=0}^{|\mathcal{C}|-1} \sum_{v=0}^{|\mathcal{V}|-1} y_{acv} rc_{vc} \tag{4.11}$$

subject to:

(a′) $Y \in \mathbb{C}_1$

(b′) $X \in \mathbb{C}_2$

(c′) $\dfrac{\sum_{a=0}^{|\mathcal{A}|-1} \sum_{c=0}^{|\mathcal{C}|-1} (\sum_{u=0}^{|\mathcal{U}|-1} \gamma_{au} x_{auc} dt_{uc} + \frac{\hat{u}}{1-\hat{u}})}{l} \leqslant m$

(d′) $u_{ac} \leqslant \hat{u} \quad \forall a \in A, \forall c \in C$

In constraint (a′), $\mathbb{C}_1$ stands for the set of feasible replica deployment plans Y that satisfy constraints (a), (b) in **Problem 1**. In constraint (b′), $\mathbb{C}_2$ refers to the set of feasible request dispatch plans X that satisfy constraints

(e), (f) in **Problem 1**. Constraint (c') is derived from constraints (c) in **Problem 1** based on eq. (4.10). Constraint (d') guarantees that the utilization rate of the VM instance for application replica R_{ac} never exceeds $\hat{u}$. With different $\hat{u}$, **Problem 2** produces a series of MILP problems that can be solved directly using popular MILP methods, e.g., cutting-plane [108] and branch and cut [116], supported by many open source software tools, e.g., Google OR-Tools [58].

4.6.2 Mixed Integer Linear Programming

Clearly increasing $\hat{u}$ can improve the overall VM utilization rate, thus decreasing TDC in **Problem 2**. Note that if $\hat{u}$ is too large, we cannot find feasible solutions to **Problem 2** subject to constraint (c'). One feasible method is to initialize $\hat{u}$ at its maximum, then repeatedly reduce $\hat{u} = \hat{u} - \Delta u$ until a feasible solution can be found. For this purpose, we theoretically analyse the upper bound on $\hat{u}$.

Let $\overline{dt}$ denote the average RTD across all requests for all application replicas. We have:

$$\overline{dt} := \frac{\sum_{a=0}^{|\mathcal{A}|-1}\sum_{c=0}^{|\mathcal{C}|-1}\sum_{u=0}^{|\mathcal{U}|-1}\gamma_{au}x_{auc}dt_{uc}}{l}. \tag{4.12}$$

Given request rate γ_{au}, $\overline{dt}$ is bounded from below by:

$$d = \frac{\sum_{a=0}^{|\mathcal{A}|-1}\sum_{u=0}^{|\mathcal{U}|-1}\gamma_{au}\min\limits_{C_c\in\mathcal{C}}\{dt_{uc}\}}{l}.$$

This lower bound is realized when all user requests are dispatched to the respective data center with the minimal dt_{uc}.

Theorem 1. *The maximum of $\hat{u}$ is $\frac{l(m-d)}{l(m-d)+|\mathcal{A}|}$ subject to constraints (c').*

Proof. Let n be the total number of application replicas, we have $n \geqslant |\mathcal{A}|$.

According to constraint (c'):

$$\begin{aligned}
\overline{dt} + \frac{n\frac{\hat{u}}{1-\hat{u}}}{l} &\leqslant m \\
\frac{n\frac{\hat{u}}{1-\hat{u}}}{l} &\leqslant m - d \\
\frac{\hat{u}}{1-\hat{u}} &\leqslant \frac{l(m-d)}{|\mathcal{A}|} \\
\hat{u} &\leqslant \frac{l(m-d)}{l(m-d)+|\mathcal{A}|}.
\end{aligned}$$

Therefore, we have proven the upper bound on $\hat{u}$. □

With $\hat{u}$, ART of any feasible solution to **Problem 2** is less than m in constraint (c'), because $u_{ac} \leqslant \hat{u}$ for all application replicas. Increasing m to m' when solving **Problem 2** is helpful to find solutions with lower TDC. However, if m' is too large, the found solutions cannot be feasible due to violation of constraint (c'), i.e., ART of the solutions exceed m. Another iterative method can be used to gradually increase $m' \leftarrow m' + \Delta m$ in constraint (c') until we cannot find feasible solutions.

Algorithm 6 shows the procedure of the MILP-based algorithm with adaptive $\hat{u}$ and m'. After initializing $\hat{u}$ (step 1), we iteratively determine an appropriate $\hat{u}$ through steps 2-4. After a feasible solution to **Problem 2** is found, we calculate ART and TDC in step 5. Next, we relax the performance requirement to m' in an attempt to decrease TDC. The process is repeated until ART is greater than m or the termination rule (e.g., the maximum optimization time) is met (step 6-14).

Finally, we theoretically analyse the performance of the base solution obtained by Algorithm 6 in terms of the worst-case ratio.

For each VM type V_v in data center C_c, we can obtain the unit cost of processing a single request for application A_a:

$$k_{acv} = \frac{rc_{vc}}{\delta_{av}}. \tag{4.13}$$

Algorithm 6 The MILP-based algorithm in MCApp.

Input: $\mathcal{A}, \mathcal{U}, \mathcal{C}, \mathcal{V}, \gamma_{au}, rc_{vc}, \delta_{a,c,v}, dt_{uc}, m$.

Output: Request dispatch plan X and replica deployment plan Y.

1: $\hat{u} \leftarrow \frac{l(m-d)}{l(m-d)+|\mathcal{A}|}$
2: **while** MILP methods cannot find a feasible solution to **Problem 2** with $\hat{u}$ **do**
3: $\quad \hat{u} \leftarrow \hat{u} - \Delta u$
4: **end while**
5: For the feasible solution X and Y to **Problem 2**, calculate ART and TDC
6: $m' \leftarrow m$
7: **while** $ART \leqslant m$ and termination rule is not met **do**
8: $\quad$ Gain X' and Y' to **Problem 2** with $m' \leftarrow m' + \Delta m$ by MILP methods and calculate its ART' and TDC'
9: $\quad$ **if** $ART' \leqslant R$ and $TDC' < TDC$ **then**
10: $\quad\quad X, Y \leftarrow X', Y'$
11: $\quad\quad TDC \leftarrow TDC'$
12: $\quad$ **end if**
13: $\quad ART \leftarrow ART'$
14: **end while**
15: **return** X and Y

We denote the minimal unit cost for application A_a across all VM types and data centers as $k_{a.min} = \min_{C_c \in \mathcal{C}, V_v \in \mathcal{V}} \{k_{acv}\}$ and the minimal unit cost among all applications as $k = \min_{A_a \in \mathcal{A}} \{k_{a.min}\}$.

Next we generate a *ARD* solution subject to all the constraints in **Problem 2**. Therefore, TDC of this solution bounds from above the TDC of the optimal solution to **Problem 2**.

Concretely, let this *ARD* solution have $\overline{dt} = d$, i.e., all user requests are served in the data centers with the minimal dt_{uc}. Based on this request dispatch plan, we can obtain the workload of application replica λ'_{ac} and

the number of replicas h. In this case, we have $\hat{u'} \leqslant \frac{l(m-d)}{l(m-d)+h}$ following the proof of Theorem 1.

Let $\hat{u'} = \frac{l(m-d)}{l(m-d)+h}$, we always choose the cheapest VM type to deploy replica R_{ac} subject to $\delta_{av'} \geqslant \frac{\lambda'_{ac}}{\hat{u'}}$. Let rc'_{ac} denote the rental cost of the cheapest VM for R_{ac}, TDC of this *ARD* solution can be calculated by:

$$e = \sum_{a=0}^{|\mathcal{A}|-1} \sum_{c=0}^{|\mathcal{C}|-1} rc'_{ac}. \tag{4.14}$$

Theorem 2. *TDC of the optimal solution to **Problem 2** is at most $\frac{e(m-d)}{kl(m-d)+k}$ times of the optimal TDC to **Problem 1**.*

Proof. We can obtain the worst-case performance ratio by comparing the upper bound on TDC of the optimal solution to **Problem 2**, i.e., TDC', and the lower bound on TDC of the optimal solution to **Problem 1**, i.e., TDC^*, respectively.

Let X^* and Y^* be the optimal request dispatch plan and replica deployment plan to **Problem 1**, respectively. Based on X^* and Y^*, we can obtain the optimal ${\lambda_{ac}}^*$ and ${\mu_{ac}}^*$ according to eq. (4.1) and (4.2). The average utilization rate of all application replicas can be calculated as follows:

$$u^* = \frac{\sum_{a=0}^{|\mathcal{A}|-1} \sum_{c=0}^{|\mathcal{C}|-1} {\lambda_{ac}}^*}{\sum_{a=0}^{|\mathcal{A}|-1} \sum_{c=0}^{|\mathcal{C}|-1} {\mu_{ac}}^*} = \frac{l}{\sum_{a=0}^{|\mathcal{A}|-1} \sum_{c=0}^{|\mathcal{C}|-1} {\mu_{ac}}^*}. \tag{4.15}$$

Let $\overline{pt}$ denote the average request processing time across all application replicas. Following eq. (4.3), we have:

$$\overline{pt} = \frac{1}{\sum_{a=0}^{|\mathcal{A}|-1} \sum_{c=0}^{|\mathcal{C}|-1} {\mu_{ac}}^* - l} = \frac{1}{l} \cdot \frac{u^*}{1-u^*}.$$

According to constraint (c) in **Problem 1**, we have:

$$\begin{aligned}\overline{dt} + \overline{pt} \leqslant & m \\ \overline{pt} \leqslant & m - d \\ \frac{1}{M} \cdot \frac{u^*}{1-u^*} \leqslant & m - d \\ u^* \leqslant & \frac{l(m-d)}{l(m-d)+1}.\end{aligned}$$

According to eq. (4.15), eq. (4.2), and eq. (4.13), we have:

$$\begin{aligned}l =& u^* \sum_{a=0}^{|\mathcal{A}|-1} \sum_{c=0}^{|\mathcal{C}|-1} {\mu_{ac}}^* = u^* \sum_{a=0}^{|\mathcal{A}|-1} \sum_{c=0}^{|\mathcal{C}|-1} \sum_{v=0}^{|\mathcal{V}|-1} \delta_{av} {y_{acv}}^* \\ =& u^* \sum_{a=0}^{|\mathcal{A}|-1} \sum_{c=0}^{|\mathcal{C}|-1} \sum_{v=0}^{|\mathcal{V}|-1} \frac{rc_{vc}}{k_{acv}} {y_{acv}}^* \\ \leqslant & \frac{u^*}{k} \sum_{a=0}^{|\mathcal{A}|-1} \sum_{c=0}^{|\mathcal{C}|-1} \sum_{v=0}^{|\mathcal{V}|-1} rc_{vc} {y_{acv}}^*.\end{aligned}$$

Therefore,

$$\begin{aligned}TDC^* =& \sum_{a=0}^{|\mathcal{A}|-1} \sum_{c=0}^{|\mathcal{C}|-1} \sum_{v=0}^{|\mathcal{V}|-1} {y_{acv}}^* rc_{vc} \\ \geqslant & \frac{kl}{u^*} \\ \geqslant & \frac{kl(m-d)+k}{m-d}.\end{aligned}$$

Finally, we can estimate the worst-case performance ratio as the following:

$$\frac{TDC'}{TDC^*} \leqslant \frac{e(m-d)}{kl(m-d)+k}.$$

We have proven the worst-case performance of the base solution found by Algorithm 6. □

4.6.3 Large Neighborhood Search

LNS has been widely used to solve various combinatorial optimization problems with practical importance [132], including multi-cloud service brokering in [68]. In this subsection, we explain the LNS-based algorithm to improve the base solutions found by Algorithm 6. LNS explores solution space by applying destroy and repair heuristics to the most recently discovered solution in each iteration [149]. Since the best achievable TDC depends on the number, locations, and types of the deployed VMs, we decide to design an LNS algorithm to improve the replica deployment plan Y. Particularly, three problem-specific heuristics are designed for replica replacement, VM selection, and request dispatching, respectively. To explore solution space, we first propose a *destroy heuristic* to remove a varying number of application replicas from Y based on a domain-tailored relatedness measurement. Next, we design a *repair heuristic* to effectively transfer the destroyed Y violating constraint (c) to a feasible Y. Upon repairing Y, we propose a *delay-oriented heuristic* for request dispatching and ART evaluation.

Algorithm 7 shows the procedure of our LNS-based algorithm. The input of Algorithm 7 X° and Y° represent the base solution found by Algorithm 6. The input parameter $iter_{max}$ denotes the maximum number of iterations without improving Y before terminating the algorithm. X_{best} and Y_{best} together denote the best *ARD* solution obtained so far during LNS. The function $destroy(Y_{best})$ in step 4 destroys a copy of Y_{best} by removing a portion of application replicas. The function $repair(Y)$ in step 5 transforms the destroyed replica deployment plan into a new feasible Y and determines the request dispatch plan X accordingly. In step 6, TDC of the new *ARD* solution is evaluated. Based on that, the solution is either rejected or accepted as the current solution for the next search iteration. Specifically, we only accept the cheaper solution and update X_{best} and Y_{best} correspondingly (step 7). Finally, Algorithm 7 returns X_{best} and Y_{best}.

Algorithm 7 The LNS-Based algorithm in *MCApp*.

Input: Base solution found by Algorithm 6, i.e., X° and Y°, $iter_{max}$.

Output: X_{best}, Y_{best}

```
1: X_best, Y_best ← X°, Y°
2: i ← 0
3: while i < iter_max do
4:    Y ← destroy(Y_best)   /* Algorithm 8
5:    X, Y ← repair(Y)      /* Algorithm 10
6:    if TDC(X, Y) < TDC(X_best, Y_best) then
7:       X_best, Y_best ← X, Y
8:       i ← 0
9:    else
10:      i ← i + 1
11:   end if
12: end while
13: return X_best, Y_best
```

Destroy Heuristic

Our *destroy heuristic* is inspired by [68], which makes large changes to the current multi-cloud service brokering solution by removing the related request assignments, e.g., the assigned services are geographically close. We propose to measure the relatedness among application replicas for destroying Y_{best}. Thus, $N(R_{ac}, R_{a'c'})$ in eq. (4.16) quantifies the *relatedness* between replicas R_{ac} and $R_{a'c'}$:

$$N(R_{ac}, R_{a'c'}) = \begin{cases} dt(C_c, C_{c'}) & \text{if } A_a = A'_a \\ \infty & \text{otherwise,} \end{cases} \tag{4.16}$$

where $dt(C_c, C_{c'})$ is the RTD between two data centers C_c and $C_{c'}$. That is, application replicas are more related if they belong to the same application and the deployed data centers C_c and $C_{c'}$ are close to each other, while less related if they belong to different applications.

Algorithm 8 Destroy heuristic.

Input: Replica deployment plan Y_{best}, ρ, r.

Output: New replica deployment plan Y after replicas removal.

1: $k \leftarrow random(1, \lceil \rho |Y_{best}| \rceil)$
2: Randomly select an application replica R_{ac} from Y_{best}
3: Initialize a set of application replicas $\mathcal{R} \leftarrow \{R_{ac}\}$
4: **while** $|\mathcal{R}| < k$ **do**
5: Create a list $\mathcal{L}$ including all application replicas from Y not in $\mathcal{R}$
6: Randomly select an application replica R_{ac} from $\mathcal{L}$
7: Sort $\mathcal{L}$ such that $i < j \Rightarrow N(R_{ac}, \mathcal{L}[i]) < N(R_{ac}, \mathcal{L}[j])$
8: Choose a random number $\xi \in [0, 1)$
9: $\mathcal{R} \leftarrow \mathcal{R} \cup \mathcal{L}[\xi^r |\mathcal{L}|]$
10: **end while**
11: Update Y_{best} as Y by removing replicas in $\mathcal{R}$
12: **return** Y

Algorithm 8 shows the procedure of the proposed destroy heuristic. The parameter $\rho \in (0, 1)$ is the percentage of application replicas to be removed and r ($r \geqslant 1$) controls the random selection of replicas for removal. To discover possibly better *ARD* solutions, our destroy heuristic first *randomly* generates an integer $k \in [1, \lceil \rho |Y_{best}| \rceil]$ as the number of replicas to be removed in step 1. Compared with removing a fixed number of replicas, our method has been verified to improve the performance of LNS in our experiments (Subsection 4.7.5). Then the heuristic randomly selects one replica R_{ac} from Y_{best} (step 2) and adds it to set $\mathcal{R}$ (step 3), which is initially empty. Next, we create a list $\mathcal{L}$ to include all application replicas not in $\mathcal{R}$ (step 5) and randomly pick up a replica R_{ac} from $\mathcal{R}$ (step 6). In step 7, the application replicas in $\mathcal{L}$ are sorted in the ascending order according to the relatedness with R_{ac} in eq. (4.16). We select one replica from $\mathcal{L}$ with a probability proposed in [68] and add the replica to $\mathcal{R}$ (step 8-9). The procedure is repeated until k application replicas have been selected for

removal. Finally, Y_{best} is updated by removing all replicas in $\mathcal{R}$.

Repair Heuristic

It is highly likely that the replica deployment plan Y after replica removal cannot satisfy constraint (c). Our *repair heuristic* aims to generate new *ARD* solutions such that $ART \leqslant m$. Particularly, we must resolve four issues during the repair process: (1) How many new replicas should be deployed? (2) Where should these new replicas be placed? (3) How to dispatch user requests based on the current replica deployment plan? (4) Which types of VMs should be deployed?

For issue (1), we *randomly* generate the required number of application replicas $g \in [|Y_{best}| - \Delta g, |Y_{best}| + \Delta g]$ to ensure that the number of replicas after repair is not too different from $|Y_{best}|$. Here, Δg is a hyper-parameter that controls the exploration of the repair heuristic. That is, a larger Δg creates more optional values for g. It is helpful to consider more different and potentially good *ARD* solutions. The method has also been verified to improve the performance of LNS in our experiments (Subsection 4.7.5).

For issue (2), i.e., identifying appropriate data centers to place new replicas, we design a *greedy*-based method to add data centers until the number of application replicas reaches g. In order to do this, we calculate each application's average utilization rate:

$$u_a = \frac{\sum_{u=0}^{|\mathcal{U}|-1} \gamma_{au}}{\sum_{c=0}^{|\mathcal{C}|-1} \mu_{ac}}. \tag{4.17}$$

The application with the highest average utilization rate is selected to receive new replicas. The rationale is as follows. On the one hand, if $u_a \geqslant 1$, for at least one application replica R_{ac}, the utilization rate $u_{ac} \geqslant 1$, which violates eq. (5). On the other hand, if $u_a < 1$, adding replicas for an application with a high average utilization rate helps to reduce the average request processing time.

Next, we determine the data center to place a new replica for the selected application. Here, we calculate the *benefit* of each data center C_c as

follows:

$$benefit_c = \frac{\overline{dt}_{a+c} - \overline{dt}_a}{k_{acv}}, \tag{4.18}$$

where $\overline{dt}_{a+c}$ and $\overline{dt}_a$ are the average RTD of application A_a after and before adding a replica for application A_a using the cheapest VM type V_v in data center C_c, respectively. The new application replica is placed in the data center with the largest benefit defined in eq. (4.18) until the number of application replicas reaches g.

Note that RTD depends on the request dispatch plan X. For issue (3), we develop a *delay-oriented* heuristic to quickly revise X for the purpose of minimizing the total RTD between users and application replicas. Here, we assume that all replicas of the same application have the identical utilization u_a, which can be calculated by eq. (4.17). Although this assumption based on the capacity-based round-robin scheduling [55] cannot guarantee the optimality of X for a given Y, it can efficiently prevent any application replica from being heavily utilized, thereby reducing the risk of long request processing time. After defining the workload, i.e., $\lambda_{ac} = u_a \mu_{ac}$, for each application replica, X is generated according to Algorithm 9.

Algorithm 9 demonstrates the process of the delay-oriented heuristic. For each application A_a, the heuristic first calculates the workload of all replicas λ_{ac} (step 2) and creates two lists, i.e., $RTDList$ and $DCList$, to record the minimal RTD among all replicas and the corresponding data center for each user region (step 3). Then the heuristic iteratively finds the user region with the minimum value in $RTDList$, and dispatches the requests from this region to the corresponding replica according to $DCList$. Concretely, if $\lambda_{ac} \geqslant \gamma_{au}$, that is, the workload of R_{ac} is in surplus, the requests rate from U_u to R_{ac}, i.e., r_{auc} is determined by γ_{au} in step 7 and λ_{ac} is updated in step 8. Otherwise, r_{auc} is determined by λ_{ac} in step 10 and the two lists $DCList$ and $RTDList$ are updated regardless of R_{ac} in step 11. Finally, the heuristic returns X by setting each of its component x_{auc} to $\frac{r_{auc}}{\gamma_{au}}$.

Algorithm 9 Delay-oriented heuristic for request dispatch.

Input: Replica deployment plan Y, γ_{au}, dt_{uc}.

Output: Request dispatch plan X.

for all Application A_a **do**
  $\lambda_{ac} \leftarrow \frac{\sum_{u=0}^{|\mathcal{U}|-1} \gamma_{au}}{\sum_{c=0}^{|\mathcal{C}|-1} \mu_{ac}} \mu_{ac}$
  Create two lists $RTDList$ and $DCList$
  while Exist requests from U_u not dispatch **do**
    Find the user region U_u with $min(RTDList)$ and the corresponding replica R_{ac} by $DCList$
    if $\lambda_{ac} \geqslant \gamma_{au}$ **then**
      $r_{auc} \leftarrow \gamma_{au}$
      $\lambda_{ac} \leftarrow \lambda_{ac} - \gamma_{au}$
    else
      $r_{auc} \leftarrow \lambda_{ac}$
      $\lambda_{ac} \leftarrow 0$ and update $DCList$ and $RTDList$ without considering R_{ac}
    end if
  end while
end for
return Request dispatch plan X

We address issue (4), i.e., VM type selection, after new replicas are placed. If the new replica deployment plan Y is not feasible, i.e., $ART > m$, we will select new VM types with higher processing speed, e.g., change the VM type from AWS m5.large to m5.xlarge, for existing application replicas.

Note that the upgrade of VMs will change request dispatch plan X and upgrading a single VM may increase the average RTD. For example, if an application has more Asian users than European users, upgrading the application replica in Europe alone will reduce the request processing time for European users. However, the upgrade can increase RTD for

Algorithm 10 Repair heuristic.

Input: Replica deployment plan Y after replicas removal.

Output: Repaired replica deployment plan Y and request dispatch plan X.

1: $g \leftarrow random(|Y_{best}| - \Delta g, |Y_{best}| + \Delta g)$
2: **while** $|Y| < g$ **do**
3: Determine the application A_a with the highest average utilization rate according to eq. (4.17)
4: Determine the data center C_c with the largest benefit according to eq. (4.18)
5: Add a new application replica for A_a at C_c with a random VM type
6: **end while**
7: Obtain request dispatch plan X by Algorithm 9 and calculate ART
8: **while** $ART > m$ **do**
9: Randomly select application replicas from Y
10: Update the selected replicas by upgrading the current VM types
11: Obtain request dispatch plan X by Algorithm 9 and calculate ART
12: **end while**
13: **return** X and Y

some Asian users due to re-dispatching requests based on Algorithm 9. To decrease ART, multiple replicas should be upgraded simultaneously. We design a *random*-based method to identify an appropriate replica set for upgrading. Progressively one application replica is randomly selected and upgraded until the ART within m. The overall repair heuristic is shown in Algorithm 10.

4.7 Evaluation of *MCApp*

In this section, we present the experimental evaluation of our proposed *MCApp* approach for solving the *ARD with flexible dispatching* problem. We

use the same datasets for the *ARD with close dispatching* problem detailed in Subsection 4.5.1.

4.7.1 Competing Approaches and Parameter Settings

MCApp is compared with *GA-ARD* and two other competing algorithms, as briefly described below.

The LNS-based approach proposed in [68] supports periodical VM selection in multi-cloud. For convenience, we denote the competing algorithm as *LNS-MC*. This approach applies a greedy-based constructive heuristic, a Shaw-based destroy heuristic [149], and a greedy-based repair heuristic. We apply the same parameter settings as [68]: $iter_{max} = 2000$, $\rho = 0.3$, $r = 4$. We attempt to fine-tune these parameters, without obtaining noticeably better performance.

HMOHM [109] is proposed to deploy Web applications to public clouds. The GA-based algorithm in *HMOHM* applies two evolutionary operators, i.e., mutation and nascency, to balance global and local search. For parameters of GA, we adopt the same settings as recommended in [109]: the population size is 100, the elite size is 10, both the mutation rate and the nascency rate are 0.5, and the termination time is 5 minutes.

We use CBC solvers from Google OR-Tools package version 7.3.7 [58] to implement our proposed MILP-based algorithm. Based on our preliminary simulation studies, we decide to set the parameters: $\Delta u = 1\%$ and $\Delta m = 5ms$. We set 10 minutes as the termination rule to control the overall computational time of *MCApp*. For our proposed LNS-based algorithm, we follow [68] to determine its parameter settings: $iter_{max} = 2000$, $\rho = 0.3$, $r = 4$. Besides, we set $\Delta g = 2$, which is sufficient to explore search space because we cannot gain significant improvements on the quality of the final solutions by using larger Δg, e.g., setting Δg as 3 or 4. To compare the results, we run each experiment 30 times on the same computer with Intel Core i7-8700 CPU (3.2 GHz and 16 GB of RAM).

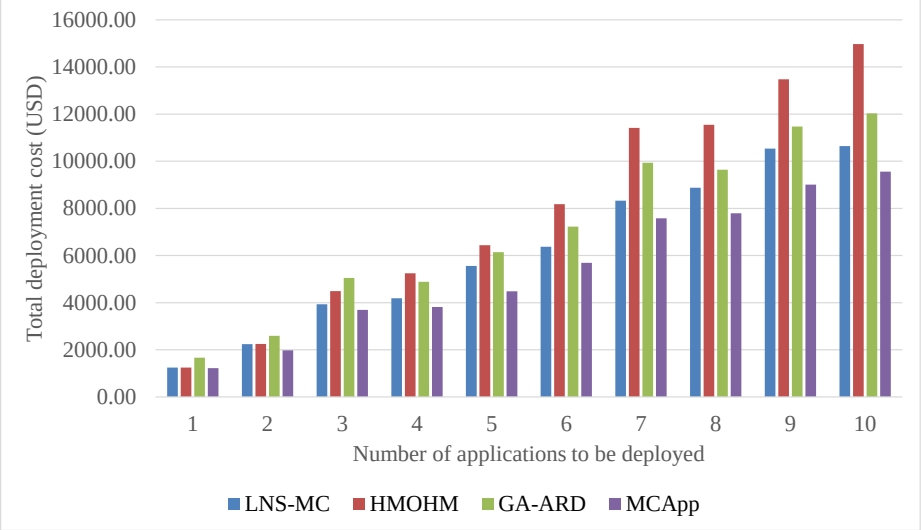


Figure 4.8: Comparison of average TDC by competing approaches.

4.7.2 Performance Comparison

Through different constraint handling techniques, all competing approaches have similar ART within 150 ms. Therefore, we show the average TDC achieved with different numbers of applications in Figure 4.8. The mean and standard deviation of TDC are presented in Table 4.3.

When the number of applications $|\mathcal{A}|$ is 1, *MCApp* outperformed *LNS-MC* and *HMOHM* by over 2%. As $|\mathcal{A}|$ increases, *MCApp* can generate solutions with much lower TDC than *LNS-MC* and *HMOHM*. Particularly, when deploying more than 4 applications simultaneously, the performance improvement of *MCApp* over *HMOHM* is substantial, i.e., the cost reduction is more than 30%. We also find that except $|\mathcal{A}| = 8$, the cost reduction of *MCApp* than *GA-ARD* is over 20%, which confirms that dispatching the requests from the same user region to different application replicas rather than the closest one results in better solutions.

From Table 4.3, the average TDC achieved by *MCApp* is 10.47% lower

Table 4.3: Performance comparison of the approaches for *ARD* with different application diversities (*TDC* per month in USD, the best is highlighted).

No. of Apps	*HMOHM* [109]		*LNS-MC* [68]		*GA-ARD*		*MCApp*	
	TDC	n^*	*TDC*	n^*	*TDC*	n^*	*TDC*	n^*
1	1246.08±10.52	2	1244.16±0	2	1661.76±0	4	**1219.68±0**	3
2	2251.7±84.4	4	2240.64±0	3	2589.6±21.62	7	**1978.56±0**	4
3	4492.32±179.33	6	3935.81±7.77	6	5049.12±121.38	17	**3694.25±41.49**	7
4	5244.1±333.92	8	4190.5±0.53	6	4881.6±24.94	13	**3816.96±30.11**	8
5	6434.93±355.3	10	5558.4±0	6	6140.16±0	17	**4481.28±0**	9
6	8182.58±587.35	12	6373.44±0	7	7227.36±0	21	**5687.16±40.48**	13
7	11413.37±819.92	14	8323.2±0	8	9933.12±0	24	**7580.52±43.15**	13
8	11549.78±709.09	16	8876.16±0	9	9644.4±19.05	29	**7794.19±197.69**	15
9	13481.4±1091.05	18	10535.04±0	10	11472.48±0	29	**9009.41±308.45**	15
10	14968.92±1421.1	20	10644.48±0	11	12034.08±6.6	34	**9558.6±237.74**	19

than *LNS-MC*, 25.55% lower than *HMOHM*, and 23.21% lower than *GA-ARD* among all problem instances. The observed performance differences between *MCApp* and three competing approaches are verified by the statistical test (Wilcoxon Rank-Sum test) with a significance level of 0.05 for all problem instances. We also find that *MCApp* has the small standard deviations, confirming its stability and reliability for the *ARD with flexible dispatching* problem.

4.7.3 Replica Amount

Next, we investigate the effectiveness of *MCApp* by comparing the numbers of application replicas in the final replica deployment plans. The replica number n^* of the *best* solution among 30 runs is presented in Table 4.3.

n^* obtained by *GA-ARD* is always greater than the other competing approaches due to the mechanism of close dispatching. In contrast, *LNS-MC* tends to use smaller n^* than *MCApp*. The smaller replica number decided

by *LNS-MC* means more expensive VMs with higher processing speed have to be rented to serve the user requests, which reduces the utilization of VMs.

MCApp and *HMOHM* achieved different n* on 8 out of 10 problem instances. From the above results, we can conclude that the number of replicas also seriously impacts TDC for *ARD with flexible dispatching*, and *MCApp* can effectively search for appropriate n for replica deployment plans. For example, when $|\mathcal{A}| = 1$, *MCApp* determines n as 3 and deploys the three application replicas in Northern Virginia, Seoul, and Mumbai data centers respectively.

4.7.4 Effectiveness of the MILP-based Algorithm in MCApp

As shown in Table 4.4, we also examine the performance of our proposed MILP-based algorithm, i.e., Algorithm 6, using two baseline methods, the greedy-based constructive heuristic in [68] (*GreedyCon* for convenience) and the heuristic-based method proposed in Subsection 4.4.3, i.e., *GreedyAdd*, in terms of TDC. The replica numbers of the base solutions n are also reported in Table 4.4.

GreedyCon achieves compatible performance as Algorithm 6 for single application deployment. However, when $|\mathcal{A}|$ is between 2 and 6, Algorithm 6 outperforms *GreedyCon* significantly. Besides, Algorithm 6 can generate better solutions than *GreedyAdd* with fewer n in all problem instances, benefiting from the flexible request dispatch plans.

For the problem instances where $|\mathcal{A}| > 6$, Algorithm 6 cannot generate better solutions than *GreedyAdd* before the termination rule is met. However, using the information obtained from the solutions generated by Algorithm 6, e.g., the appropriate replica number for applications, *MCApp* still can obtain high-quality final solutions by our proposed LNS-based algorithm, i.e., Algorithm 7. Overall, Algorithm 7 improves the base solutions from Algorithm 6 for all 10 problem instances, whereas *GreedyCon* so-

Table 4.4: Base solutions' comparison for deploying applications with different application diversities (TDC per month in USD, the best is highlighted).

No. of Apps	Algorithm 6		*GreedyCon* [68]		*GreedyAdd*	
	TDC	n	TDC	n	TDC	n
1	**1244.16**	2	**1244.16**	2	1716.48	5
2	**2052.00**	5	2240.64	3	2602.08	8
3	**3767.04**	8	4193.28	6	5119.2	13
4	**4033.44**	9	4193.28	6	4910.4	12
5	**5353.92**	11	5558.40	6	6140.16	17
6	**6134.40**	12	6373.44	7	7227.36	21
7	9385.92	13	**8323.20**	8	9933.12	24
8	9650.88	17	**8876.16**	9	9666	28
9	11269.44	17	**10535.04**	10	11472.48	29
10	11721.60	19	**10644.48**	11	12078.72	32

lutions can only be improved for 2 out of 10 problem instances ($|\mathcal{A}| = 3, 4$). Therefore, the hybrid approach *MCApp* is effective in finding good *ARD* solutions.

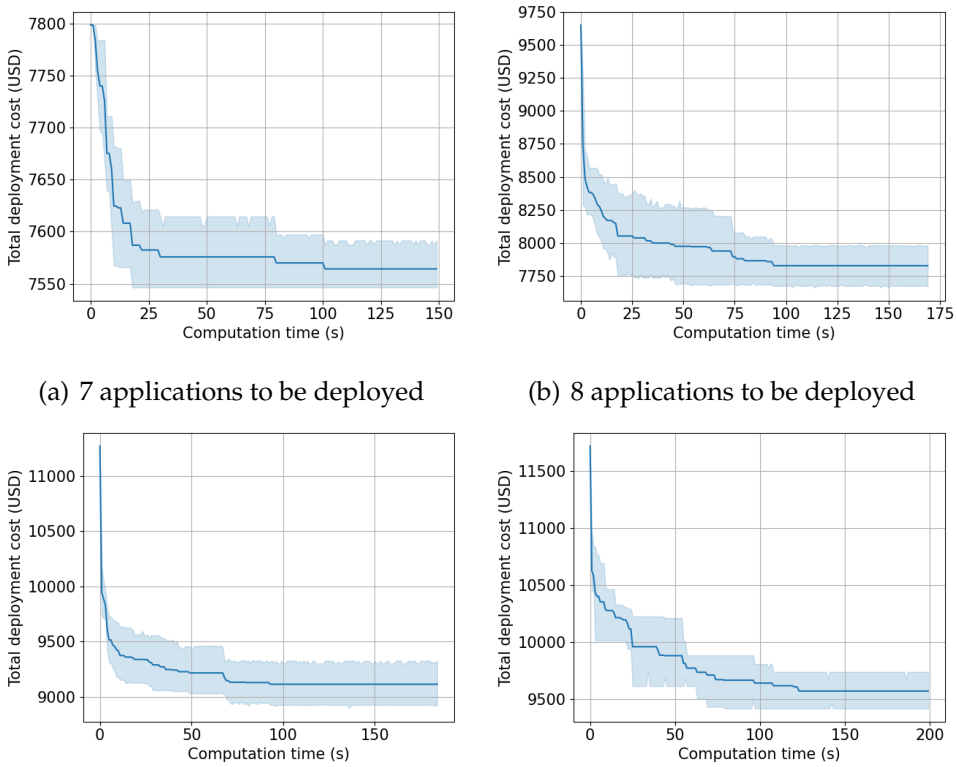
4.7.5 Effectiveness of the LNS-Based algorithm in MCApp

Since there are several newly developed heuristics in our LNS-Based algorithm, i.e., Algorithm 7, we perform ablation studies to analyse its effectiveness. For the destroy heuristic, we compare Algorithm 8 with the destroy heuristic in [68] with a fixed $k = \lceil \rho |Y_{best}| \rceil$. For the repair heuristic, we compare Algorithm 10 with the repair heuristic in [68] with a fixed $g = |Y_{best}|$. For application replica placement, we compare our greedy method with the random method. For VM type selection, we compare our random method with the greedy method. The mean TDC of these ablation heuristics are presented in Table 4.5.

Table 4.5: Ablation of Algorithm 7 across each contribution in mean TDC

No. of Apps	Fixed k	Fixed g	Random placement	Greedy selection
1	1219.68	1244.16	1232.54	1244.16
2	1979.14	1995.84	1978.66	1978.56
3	3724.75	3684.96	3757.44	3767.04
4	3877.56	3815.83	3918.50	4021.92
5	4484.78	4488.12	4483.34	4492.51
6	5732.11	5606.21	5720.74	5714.64
7	7798.32	7610.81	7642.32	7664.40
8	8779.99	7860.34	8164.97	8190.94
9	10043.30	8983.90	9498.12	9657.65
10	10733.28	9658.63	10218.10	10419.07

First, we can observe that Algorithm 8 achieves better performance than [68] with the fixed k (shown as Fixed k in Table 4.5) except for the single application deployment. This means that removing a random number of replicas in step 2 of Algorithm 8 helps to explore search space, especially for high application diversity (the improvements exceed 10% when $|\mathcal{A}| = 8, 9, 10$). Second, setting g randomly in step 1 of Algorithm 10 to encourage exploration can lead to better performance than the fixed g in [68] (shown as Fixed g in Table 4.5) for the majority of problem instances ($|\mathcal{A}| = 1, 2, 5, 7, 8, 10$). Third, the greedy-based method for placing new replicas in steps 3 and 4 of Algorithm 10 performs significantly better than the random method (shown as Random placement in Table 4.5). This supports our analysis in Subsection 4.3.2, e.g., replicating the application with the highest average utilization rate. Lastly, the random-based method for VM selection in steps 5 and 9 of Algorithm 10 clearly outperformed the greedy method (shown as Greedy selecting in Table 4.5). The random method works effectively in the scenario where the upgrade of a single



(a) 7 applications to be deployed

(b) 8 applications to be deployed

(c) 9 applications to be deployed

(d) 10 applications to be deployed

Figure 4.9: Changes of TDC of Algorithm 7

VM has an indeterminate impact on ART as analysed in Subsection 4.3.2.

We depict the change of TDC obtained by Algorithm 7 in Figure 4.9. While the experiment results are obtained on the problem instance with 7, 8, 9, and 10 applications to be deployed, the same convergence behavior has also been witnessed on other problem instances. In Figure 4.9, the convergence of TDC can be found after the computation time of Algorithm 7 exceeds about 100-130s.

4.7.6 Further Analysis

In this subsection, we analyse the overhead of our proposed approach, i.e., *MCApp*. Since *MCApp* combines the iterative MILP and the domain-

Table 4.6: Variable numbers, numbers of constraints, and overhead of *MCApp* (CT and TCT in s.)

No. of Apps	Variable No. of X	Variable No. of Y	No. of constraints	CT of Alg. 1	CT of Alg. 2	TCT of *MCApp*
1	450	120	61	1.45	14.26	15.71
2	900	240	121	183.9	21.75	205.65
3	1350	360	181	296.47	40.87	337.34
4	1800	480	241	468.87	52.73	521.6
5	2250	600	301	600.00	55.23	655.23
6	2700	720	361	600.00	133.98	733.98
7	3150	840	421	600.00	146.06	746.06
8	3600	960	481	600.00	169.57	769.57
9	4050	1080	541	600.00	183.3	783.3
10	4500	1200	601	600.00	196.51	796.51

tailored LNS, the total overhead of *MCApp* includes the Computation Time (CT) of Algorithm 6 and Algorithm 7. In Table 4.6, we present the observed CT of the two algorithms with respect to different numbers of applications. The corresponding number of variables (i.e., x_{auc} and y_{acv}) and number of constraints (i.e., (a'), (b'), and (d') in **Problem 2**) are also included in Table 4.6. We find that the Total Computation Time (TCT) of *MCApp* increases with the number of applications to be deployed because every application will produce some variables and constraints to be handled by *MCApp*. As shown in Table 4.6, Algorithm 7 spends most of TCT when $|\mathcal{A}| = 1$. For $|\mathcal{A}| > 1$, CT of Algorithm 6 takes up a majority of TCT. The increasing number of variables and constraints has a greater impact on CT of Algorithm 6 than Algorithm 7. Overall, *MCApp* can generate a *ARD* solution within 15 minutes. The competing approach *LNS-MC* can generate solutions within 10 minutes. We attempt to extend the termination time of *HMOHM*, i.e., to 15 minutes, without obtaining noticeably

better performance. The computation time required by *GA-ARD* varies from 20 to 1500 seconds subject to the number of applications.

As confirmed by many existing studies [23], [135], the workload of many applications does not change significantly within an hour. *MCApp* can be periodically applied to re-optimize the replica deployment plan and request dispatch plan, e.g., every hour, to reduce the total deployment cost while satisfying the constraint on average response time.

4.8 Chapter Summary

In this chapter, we study the *ARD* problem in multi-cloud to satisfy the requirements on low average response time. With two types of request dispatching, we propose two approaches to minimize the total deployment cost.

To solve the *ARD with close dispatching* problem, we propose *GA-ARD* to optimize the replica placement and VM selection in multi-cloud for application replicas. Due to the complexity of the problem, we design new problem-specific solution representation, fitness measurement, and population initialization, which are effective for breeding excellent offspring. The experiments based on the datasets collected from real-world cloud providers, network environments, and applications show that *GA-ARD* outperforms *H-GA* and the common application replication and placement strategies.

To solve the *ARD with flexible dispatching* problem, *MCApp* optimizes both replica deployment and the request dispatching. *MCApp* first transforms the nonlinear *ARD* problem into a series of MILP problems by bounding the utilization rate of VMs for application replicas. Further, we propose a MILP-based algorithm to effectively generate a base solution with good resource utilization. Furthermore, to further explore the solution space, we design a problem-specific LNS-based algorithm to optimize the base solution. Our experiments show that *MCApp* can achieve up to 25%

reduction in TDC compared with *GA-ARD* and recently developed approaches.

Our experiments show that both *GA-ARD* and *MCApp* require fast computing facilities to function effectively. For some applications with highly dynamic workloads, machine learning-based approaches are required to realize elastic deployment in real-time [79]. Next chapter will study elastic application deployment (*EAD*) in multi-cloud. Different from *CAD* and *ARD*, *EAD* relies on containers to realize the run-time management of applications [18], [140].

Chapter 5

Elastic Application Deployment

5.1 Introduction

This chapter addresses Elastic Application Deployment (*EAD*) in multi-cloud. The elastic application deployment should dynamically acquire and release resources to handle varying workloads of applications. The elasticity of application deployment can improve cost-effectiveness for application providers [201]. Currently, elastic development and run-time management of applications increasingly rely on containers, an industry-leading lightweight virtualization technology [193]. By bundling together an application with all its dependencies (e.g., libraries and code), the containerized application can realize fast deployment and migration in clouds [18], [140].

Containers lay the technical foundation for scalable application deployment through *vertical scaling and horizontal scaling*. The vertical scaling changes the container configuration (i.e., the amount of granted resources) for application replicas with practically no downtime [123]. The horizontal scaling changes the number of containers for application replicas, i.e., containerized applications[1]. When adding application replicas, the time

[1]In the remainder of this chapter, we use application replicas, containerized applications, and containers interchangeably.

Table 5.1: Unit cost (vCPU per hour) of different cloud providers across different regions

Cloud Providers	Min Cost (Regions)	Max Cost (Region)
Amazon	$0.04048 (N.Virginia, Ohio, Oregon, and Ireland)	$0.0696 (Sao Paulo)
Microsoft	$0.0405 (East/North Central US and North Europe)	$0.08101 (Brazil South)
Google	$0.0445 (Lowa, Oregon, and South Carolina)	$0.0707 (Sao Paulo)

needed to launch the new containers must be considered. In multi-cloud, the horizontal scaling is achieved by geographically distributed containers from different cloud providers. Therefore, it has the advantage to address the workload changes in terms of geographical distribution. For the regional workload variations, vertical scaling is favored due to its fast response speed. Note that the prices of resources assigned to containers in different regions can vary substantially. Table 5.1 shows the minimum and maximum unit prices of vCPU across all data centers of three leading cloud providers. We observe that the most expensive region can incur up to twice the cost compared to the cheapest one. For example, the price of Microsoft in Brazil South is $0.08101, while the price of Amazon in N.Virginia, Ohio, Oregon, and Ireland is $0.04048.

In practice, enterprise application providers usually apply threshold-based rules to scale the number and/or capacity of containers [13]. This strategy is efficient to make scaling decisions at run-time. However, manually choosing appropriate thresholds is difficult and often results in containers either under-utilized that waste money or over-utilized that slow

down the request processing speed [79]. Besides, most existing studies for *EAD* focus on auto-scaling techniques within a single data center to handle the dynamic workload of applications [40], [141], [177], [217]. To find the most cost-effective deployment subject to the constraint on the average response time for containerized applications, we should consider geographically distributed data centers in multi-cloud. However, considering both the various configurations and locations of containers involves a complex search space. The existing approaches for *EAD* in multi-cloud suffer the high computational cost to obtain adaptive solutions. For example, the GA-based approach proposed in [6] spends about 6 minutes to generate deployment solutions. Because the start-up and shut-down time of containers are short, fast scaling decisions are expected [213]. Moreover, these approaches usually do not consider the impact of current deployment decisions on the future workload variations, which is important when we minimize the total deployment cost over a time span such as a billing day in practice.

In this chapter, we study the *EAD* problem by learning a *policy*, as an efficient strategy to decide when and what scaling actions should be performed so that the total deployment cost over a billing day is minimized while satisfying the constraint on average response time.

Deep reinforcement learning (DRL) allows to express *what* an application provider aims to obtain, instead of *how* it should be obtained [141]. The nature of DRL makes it very appealing to adaptively scale containers for applications with dynamically changing and widely distributed workloads. For example, a deep Q-network (DQN) can be trained to approach an optimal policy for container scaling to minimize the cumulative cost in the long run [29], [100], [207]. However, it is challenging to achieve both cost-effectiveness and constraint satisfaction by directly using DQN [213]. Therefore, new learning techniques must be developed to effectively solve the *EAD* problem.

Predictive strategies such as time series analysis have been used to im-

prove the timeliness of scaling decisions, which is important to achieve cost-effective application deployment [213]. Recurrent neural networks (RNNs) are one of the most popular machine learning models for time series prediction [134]. As an advanced technology for building RNNs, long short-term memory (LSTM) is capable of identifying complicated temporal dependencies and patterns in complicated time series data [161]. In this chapter, we seek to utilize an LSTM neural network to provide additional workload information for making well-informed scaling decisions.

The performance constraint of the *EAD* problem motivates us to adopt safe reinforcement learning (RL), which aims to learn a policy that maximizes the expected return, while also ensuring the satisfaction of some safety constraints during the entire learning process [30]. In the literature, there are two main approaches for safe RL: (1) augmenting the optimization criterion using penalty methods [50], [158], and (2) adopting safe exploration [49]. In this chapter, we apply a combination of the two approaches to increase the chance of satisfying the constraint on the average response time. On the one hand, we propose a penalty-based reward function to train policies toward the constraint-compliant scaling. On the other hand, we design a safety-aware action executor to ensure that any scaling decisions made by DRL will not prolong the average response time beyond the acceptable level. The main contributions of this chapter are summarized as follows.

Firstly, we identify and formulate the *EAD* problem with the objective to minimize the total deployment cost subject to the constraint on the average response time across widely distributed user communities. To the best of our knowledge, this is the first study in the literature on the elastic containerized application deployment considering both the vertical scaling and the location-aware horizontal scaling in multi-cloud.

Secondly, we propose a novel DRL-based approach, namely *DeepScale*, with an LSTM-based prediction model for the *EAD* problem. *DeepScale* features the newly designed safety-aware action executor and penalty-

based reward function.

Finally, we develop a prototype of *DeepScale* using PyTorch [130], which supports three leading cloud providers, i.e., Amazon, Microsoft, and Google. We evaluate the effectiveness of *DeepScale* through extensive experiments using realistic workloads for Web applications, i.e., WikiBench [184]. The experiment results show that *DeepScale* achieves up to 23% savings in terms of the deployment cost, compared to Amazon auto-scaling service [13] and the recently proposed baselines such as *A-SARSA* [213]. Moreover, *DeepScale* can achieve 100 percent constraint satisfaction for different applications.

5.2 Chapter Organization

The remainder of this chapter is organized as follows. Section 5.3 presents the *EAD* problem formulation, including the vertical scaling and the location-aware horizontal scaling in multi-cloud. In Section 5.4, we present our approach *DeepScale*. In Section 5.5, we describe the prototype setting and present the experimental results. We conclude this chapter in Section 5.6.

5.3 *EAD* Problem Formulation

The aim of the *EAD* problem is to scale containers for an application under a dynamically changing and distributed workload to minimize the total deployment cost over a time span subject to the constraint on average response time. The important notations are summarized in Table 5.2.

In practice, an application involves a potentially large and dynamically changing number of requests from widely distributed users in global user regions $\mathcal{U}$. Suppose that the entire time span, e.g., a billing day, is divided into fixed-size execution periods. We denote the t-th period as I_t. During time period I_t, we represent the workload from user region U_u ($U_u \in \mathcal{U}$) in

Table 5.2: Mathematical notations

Notation	Definition
I_t	The t^{th} time period
U_u	The u^{th} user center
$\gamma_u(t)$	Application request rate from user region u during I_t
$\omega(t)$	The application workload during I_t
C_c	The c^{th} data center
$A_c(t)$	The application replica deployed in C_c during I_t
rc_c	The unit cost of vCPUs for containers in C_c
$x_c(t)$	Number of vCPUs provisioned to $A_c(t)$ during I_t
$CPU(t)$	Container deployment plan during I_t
$DC(t)$	Application deployment cost during I_t
$pt_c(t)$	Average request processing time of $A_c(t)$ during I_t
$\mu_c(t)$	Workload of $A_c(t)$ during I_t
$\lambda_c(t)$	Capacity of $A_c(t)$ during I_t
dt_{uc}	Round-trip delay between user region U_u and data center C_c
$ART(t)$	Average response time of application requests during I_t
m	Maximum average response time per application request
$\sigma_{u,c}(t)$	Percentage of requests from user region u to a_c during I_t
$u(t)$	Average CPU utilization of containers during I_t
$ANL(t)$	Average network latency between user regions and application replicas during I_t

terms of application request rate as $\gamma_u(t)$. The application workload during I_t, i.e., $\omega(t) = \sum_{u=0}^{|\mathcal{U}|-1} \gamma_u(t)$. Multiple application replicas can be utilized in parallel to process the incoming requests. Each replica works independently and processes a subset of the incoming requests [140]. Similar to [140], we adopt a hierarchical architecture to manage all containerized applications scalably, following the master-workers pattern [196].

We consider a set of multi-cloud data centers $\mathcal{C}$. In data center $C_c \in \mathcal{C}$, a collection of resources can be allocated to the containerized application. There are usually four main kinds of resources in public clouds: vCPU,

memory, storage, and bandwidth. Since the computing resource is the main factor, we can assume that there are sufficient memory, storage, and network capacity for the containerized application [64], [173]. Let $A_c(t)$ denote the containerized application to be deployed in data center C_c during time period I_t and $x_c(t)$ ($x_c(t) \in \mathbb{Z}^+$) denote the number of vCPUs provisioned to the application replica $A_c(t)$. The application deployment plan during I_t can be completely captured by the vCPU provision vector $CPU(t) = [x_c(t)]_{C_c \in \mathcal{C}}$, including the number of vCPUs provisioned in all multi-cloud data centers.

Let $\sigma_{u,c}(t) \in [0, 1]$ denote the percentage of requests from user region U_u to application replica $A_c(t)$ during time period I_t. The *workload* of $A_c(t)$ can be measured by:

$$\lambda_c(t) = \sum_{u=0}^{|\mathcal{U}|-1} \gamma_u(t)\sigma_{u,c}(t). \tag{5.1}$$

Note that we have $\sum_{c=0}^{|\mathcal{C}|-1} \sigma_{u,c}(t) = 1$, that guarantees that any application request from user regions will be processed.

Let $\mu_c(t)$ denote the *capacity* of $A_c(t)$, i.e., the maximum amount of application requests processable by $A_c(t)$ per time unit. Following [187], we model the operation of each individual containerized application as an $M/M/1$ queue. According to Little's Law [98], the average request processing time of $A_c(t)$ depends on both $\mu_c(t)$ and $\lambda_c(t)$:

$$pt_c(t) = \frac{1}{\mu_c(t) - \lambda_c(t)}, \tag{5.2}$$

where $\mu_c(t) > \lambda_c(t)$ guarantees that the workload of $A_c(t)$ will not exceed its capacity.

Let dt_{uc} represent Round-Trip Delay (RTD) between user region U_u and data center C_c, we can calculate the average response time for all the user requests during time period I_t by:

$$ART(t) = \frac{\sum_{c=0}^{|\mathcal{C}|-1} \sum_{u=0}^{|\mathcal{U}|-1} \gamma_u(t)\sigma_{u,c}(t)(dt_{uc} + pt_c(t))}{\omega(t)}. \tag{5.3}$$

Let rc_c denote the unit cost of vCPUs for containers in data center C_c. Based on the application deployment plan during I_t, i.e., $CPU(t)$, the deployment cost of all application replicas during I_t can be calculated by:

$$DC(t) = \sum_{c=0}^{|\mathcal{C}|-1} x_c(t) rc_c. \tag{5.4}$$

The total deployment cost is the cumulative deployment cost over the time span with T period, i.e., $t \in \{1, ..., T\}$. Therefore, the *EAD* problem can be formulated as follows:

$$\min \quad \sum_{t=1}^{T} DC(t), \tag{5.5}$$

subject to:

$$\frac{\sum_{t=1}^{T} ART(t)\omega(t)}{\sum_{t=1}^{T} \omega(t)} \leqslant m. \tag{5.6}$$

Constraint (5.6) guarantees that the average response time over the entire time span is below the acceptable threshold m set by the application provider.

5.4 *DeepScale* for *EAD*

Our proposed approach *DeepScale* solves the *EAD* problem through scaling containers both horizontally and vertically for application deployment in multi-cloud. In this section, we present *DeepScale*. We start with a high-level overview of *DeepScale* and then describe the details on how we achieve the self-adaptive scaling.

5.4.1 Overview of *DeepScale*

With sequential decision-making under uncertainty, EAD is naturally an RL problem. *DeepScale* realizes auto-scaling by a DRL-based policy. The scaling policy decides when and what scaling actions are performed to minimize the total deployment cost subject to the constraint on average response time. To improve the timeliness and accuracy of scaling actions, *DeepScale* includes an LSTM-based workload predictor, which is effective for predicting workloads of cloud applications [91]. Figure 5.1 illustrates the overview of *DeepScale*.

In the *training phase*, *DeepScale* trains the LSTM-based workload predictor based on request arrival history. Then the DRL-based scaling policy is trained considering the predicted future workload and the monitoring of containers (i.e., resource utilization [140]) from the current deployment environment. In the *execution phase*, the trained workload predictor and scaling policy are commissioned to scale containers for incoming application requests. Before each time period I_t, *DeepScale* first predicts the application workload during I_t using the workload predictor. Based on the predicted workload and the current monitoring of containers, *DeepScale* performs scaling for time period I_t by the scaling policy.

Because different containers may have largely different capacities in terms of vCPU numbers, *DeepScale* applies Capacity-based Weighted Round-Robin (CWRR) [76] to dispatch requests among all application replicas. That is, the percentage of requests from user region U_u to application replica $A_c(t)$, i.e., $\sigma_{u,c}(t) \propto x_c(t)$. The rationale of CWRR request dispatching is two-fold. On the one hand, CWRR is commonly used in practice due to its simplicity and low computational cost [144]. On the other hand, with CWRR, all user requests tend to be dispatched to containers with large capacities. Thus it can prevent any application replica from being heavily utilized, thereby reducing the risk of long queuing time. Next, we describe the details on how *DeepScale* trains the LSTM-based workload predictor and DRL-based scaling policy.

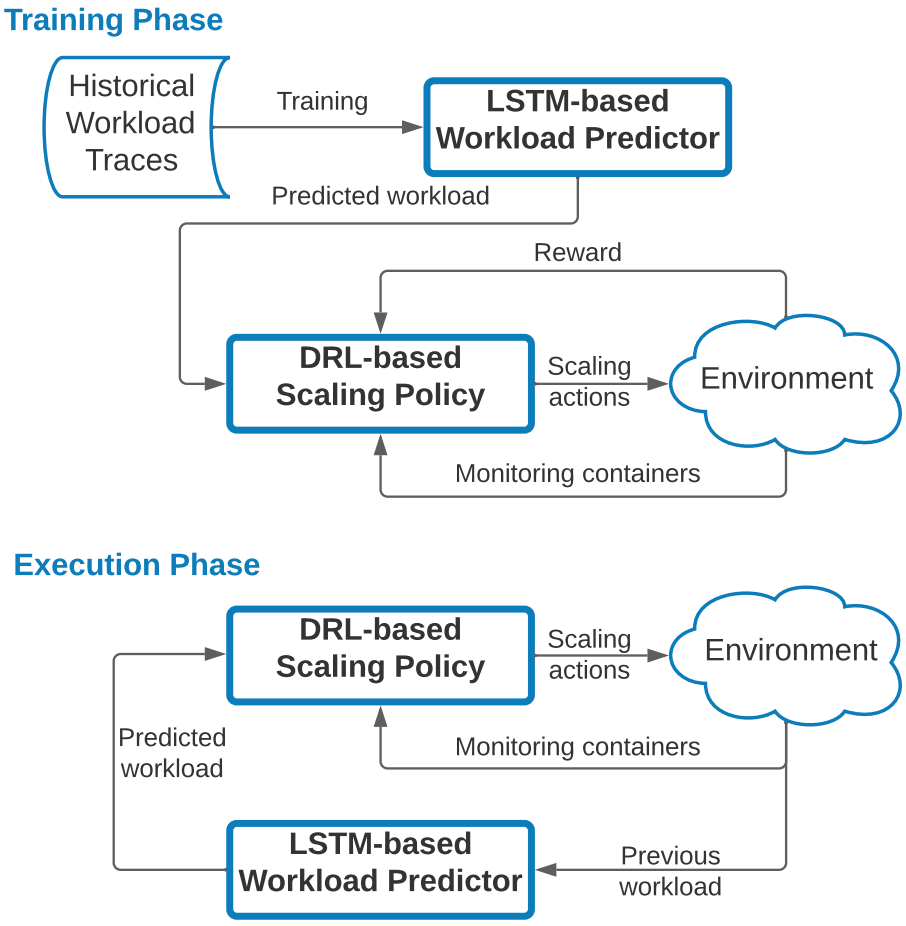


Figure 5.1: Overview of *DeepScale*.

5.4.2 Training the LSTM-based Workload Predictor

Because the application workload in terms of request rate is the key information for scaling containers, we use an LSTM neural network to obtain the predicted workload $w(t+1)$ before I_{t+1}. The prediction neural network has an input layer with 10 nodes to receive previous workloads in the last 10 time periods, i.e., $w(t), w(t-1), ..., w(t-9)$, a hidden layer with 30 LSTM units, and an output layer with 1 node to predict the workload in the next time period, i.e., $w(t+1)$. The network architecture can provide

excellent performance in predicting future workloads on the request trace in our experiments. We use Mean Square Error (MSE) as the loss function and Adam [90] as the optimizer for training the LSTM neural network. In our experiments, the LSTM neural network always converges within 100 episodes.

Note that LSTM is sensitive to the scale of the input data. Based on the historical maximum and minimum workloads, we normalize the workload data to the range between 0 and 1 when training the LSTM neural network. Afterward, the output data of the neural network, i.e., the predicted workload, is transformed back to the original scale.

5.4.3 Training the DRL-based Scaling Policy

We design a DRL-based scaling system for the *EAD* problem and define the scaling system as follows.

- **State:** The observed state includes the current container deployment plan (discrete), resource utilization (continuous), and the predicted workload (continuous).
- **Action:** To perform scaling actions (discrete), i.e., to adjust the number of containers (horizontal scaling) and/or the number of vCPUs provisioned to current containers (vertical scaling).

When performing vertical and horizontal scaling in multi-cloud, the number of potential scaling actions is indeterminate. Fixed-size action space is difficult to determine. An intuitive method is to let DQN make a *high-level* scaling decisions, i.e., to increase (*scale-up*), decrease (*scale-down*), or *maintain* the *total* number of vCPUs provisioned to containerized applications. To further decide concrete horizontal and/or vertical scaling actions, we design an action executor to make *low-level* scaling decisions based on problem-tailored heuristics. For example, after a *scale-up* high-level scaling decision is made, the low-level scaling decision will add one

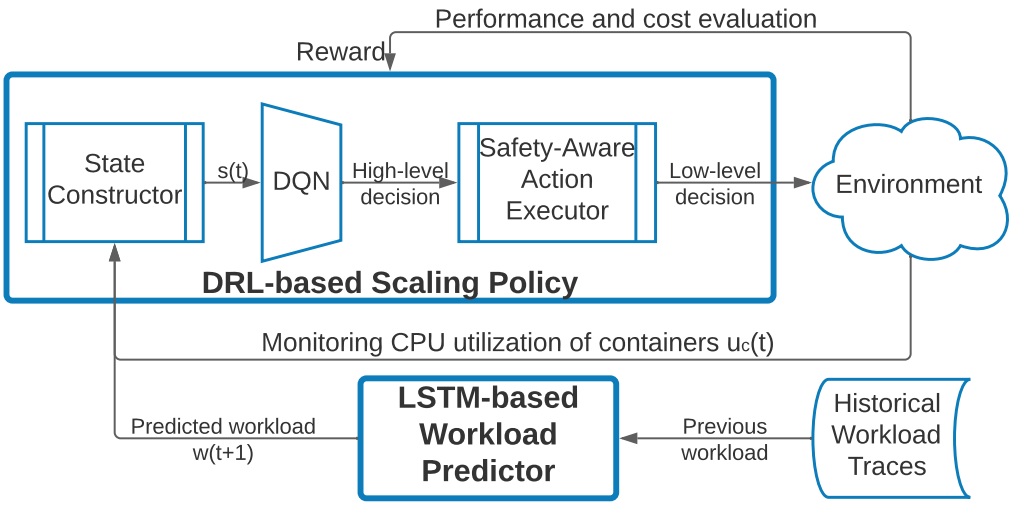


Figure 5.2: Training DRL-based Scaling Policy.

unit of vCPU to one current container and/or launch a new container with one vCPU unit.

Another DRL design challenge is to meet constraint (5.6). To this end, we propose a penalty-based reward function to guide constraint-aware Q-learning. The proposed action executor applies safe exploration to ensure that any scaling decisions made by DRL will not deteriorate the application performance. With the two mechanisms, *DeepScale* can effectively satisfy the constraint on the average response time.

Figure 5.2 shows the DRL-based scaling policy, which is composed of a *State Constructor*, a *DQN*, and a *Safety-aware Action Executor*. In the following, we provide a detailed description of each component.

State Constructor

Referring to the centralized architecture for geo-distributed and elastic deployment of containers in Kubernetes, i.e., ge-kube [140], the monitoring information about the CPU utilization of containers can be periodically

collected through RESTful APIs, e.g., the Metrics API in Kubernetes[2]. Let $u_c(t)$ denote the average CPU utilization of the container for $A_c(t)$ during I_t. At the end of time period I_t, the state constructor calculates the average CPU utilization of all containers by: $u(t) = \frac{\sum_{c:x_c(t)>0} u_c(t)}{n(t)}$, where $n(t)$ is the number of containers. The container deployment plan, i.e., $CPU(t)$, and $u(t)$ are considered as state features because they significantly affect the deployment cost and average response time of containerized applications.

Next, the state constructor adopts the change between the predicted future workload and the current workload, i.e., $\Delta w(t) = w(t+1) - w(t)$, as the state feature, because it is more straightforward for DQN to make high-level scaling decisions, i.e., changing the total number of provisioned vCPUs. Note that in both the training phase and the execution phase, the future workload $w(t+1)$ should be predicted based on the historical traces of application requests. To sum up, the output of state constructor is $s(t) = [CPU(t), u(t), \Delta w(t)]$.

DRL for Training DQN

The reward function is designed to guide DRL to minimize the total deployment cost over time span subject to the constraint on average response time:

$$r(s(t), a(t)) = -DC(t) - \max(0, (ART(t) - m)). \tag{5.7}$$

We apply Q-learning to maximize value function $Q(s, a)$, which estimates the accumulative value. The DRL-based scaling policy applies a DQN in Figure 5.2 as the function approximator. Following many existing research works [86], [207], we use experience replay [118] to stabilize Q-learning. The detailed procedures are shown in Algorithm 11.

[2]https://kubernetes.io/docs/tasks/debug-application-cluster/resourcemetrics-pipeline/.

Algorithm 11 Training DQN

Initialize: Experience replay memory $\mathcal{D}$

Output: DQN parameters

for $episode = 1\ to\ Max\_Episode$ **do**
  for $t = 1\ to\ T$ **do**
    Obtain current state $s(t)$ from the state constructor
    With probability ϵ select a random action, otherwise select an action with maximum Q-value
    Perform container scaling using the chosen action $a(t)$ through the action executor
    Observe state transition with new state $s(t+1)$, calculate $r(s(t), a(t))$ based on eq. (5.7)
    Store transition $(s(t), a(t), r(s(t), a(t)), s(t+1))$ in $\mathcal{D}$;
    Updating $Q(s(t), a(t))$
  end for
  Update DQN parameters using new Q-value estimates
end for

Safety-aware Action Executor

We design a safety-aware action executor for scaling containers in multi-cloud. Based on high-level decisions from DQN, i.e., to scale-up, scale-down, or maintain the total vCPU number, the action executor makes low-level scaling decisions as follows.

Scale-up: When the major decision to scale-up vCPUs is made, the action executor will increase the total number of provisioned vCPUs by at least one unit. Particularly, the *benefit* of each multi-cloud data center C_c is calculated by:

$$benefit_c^+ = \frac{ANL(t) - ANL_c^+(t)}{DC_c^+(t) - DC(t)}, \tag{5.8}$$

where $ANL_c^+(t)$ and $DC_c^+(t)$ are the new average network latency and deployment cost after increasing one vCPU unit in C_c. Particularly, we cal-

Algorithm 12 Action executor scale-up vCPUs

Input: General decisions to scale-up vCPUs.

Output: The specific scaling actions.

1: $Termination \leftarrow False$
2: **while** $Termination = False$ **do**
3: Calculate the benefit for each multi-cloud data center $benefit_c$ based on eq. (5.8)
4: Add one vCPU unit to the container or launch a new container with one vCPU unit in the data center having the largest $benefit_c^+$
5: Evaluate the new average CPU utilization $u'(t)$ based on eq. (5.10)
6: **if** $u'(t) < 1$ **then**
7: $Termination \leftarrow True$
8: **end if**
9: **end while**

culate the average network latency by:

$$ANL(t) = \frac{\sum_{c=0}^{|\mathcal{C}|-1} \sum_{u=0}^{|\mathcal{U}|-1} \gamma_u(t) \sigma_{u,c}(t) dt_{uc}}{\omega(t)}. \tag{5.9}$$

Note that for the current application deployment plan, i.e., $CPU(t) = [x_c(t)]_{C_c \in \mathcal{C}}$, if $x_c(t) > 0$, the one vCPU unit will be added to the current container in C_c (vertical scaling). Otherwise, a new container with one vCPU unit should be launched in C_c (horizontal scaling). Based on eq. (5.8), a larger $benefit_c$ means increasing one vCPU unit in C_c can improve performance with lower cost. Therefore, the action executor chooses the data center with the largest benefit to perform the vertical scaling or horizontal scaling. Next, we introduce a safe mechanism to ensure that a sufficient number of vCPUs can be increased in one scaling action. Particularly, the new average CPU utilization after increasing one vCPU unit is estimated by:

$$u'(t) = \frac{w(t+1)}{\sum_{c \in C} \mu_c(t)}, \tag{5.10}$$

Algorithm 13 Action executor to scale-down vCPUs

Input: General decisions to scale-down vCPUs.

Output: The specific scaling actions.

1: $Termination \leftarrow False$
2: **while** $Termination = False$ **do**
3: Evaluate the new average CPU utilization $u'(t)$ if one vCPU unit is reduced based on eq. (5.10)
4: **if** $u'(t) < 1$ **then**
5: Reduce one vCPU unit from the container with the largest $benefit_c^-$ defined in eq. (5.11)
6: **else**
7: $Termination \leftarrow True$
8: **end if**
9: **end while**

where $w(t+1)$ is the predicted workload during the next time period. If $u'(t) \geqslant 1$, one more vCPU unit is increased following the above process. Otherwise, the action executor stops increasing vCPUs. The safety mechanism aims to handle the situation when there is a surge in workload. Algorithm 12 demonstrates the overall steps of the proposed action executor for increasing vCPUs.

Scale-down: When the high-level decision to scale down vCPUs is made, we introduce another safety mechanism to avoid the potential deterioration of application performance. Particularly, the action executor first estimates the average CPU utilization $u'(t)$ after reducing one vCPU unit from current containers by eq. (5.10). If $u'(t) < 1$, the action executor reduces the vCPU unit from the container with the largest *benefit* calculated by:

$$benefit_c^- = \frac{DC(t) - DC_c^-(t)}{ANL_c^-(t) - ANL(t)}, \tag{5.11}$$

where $ANL_c^-(t)$ and $DC_c^-(t)$ are the new average network latency and deployment cost after decreasing one vCPU unit in C_c. If $u'(t) \geqslant 1$, the ac-

tion executor aborts decreasing vCPUs to prevent containers from being heavily utilized and undesired response delay. Algorithm 13 presents the overall process to decrease vCPUs.

Maintain: When the high-level decision of maintaining vCPUs is made, the action executor will attempt to reduce deployment cost and improve application performance simultaneously. Firstly, the new $ART(t)$ and $DC(t)$ are estimated by increasing one vCPU unit to the container with the largest benefit and reducing one vCPU unit from the container with the smallest benefit sequentially. Only when both the estimated $ART(t)$ and $DC(t)$ decrease, the action executor acts on the low-level scaling decision. The safety mechanism will avoid reconfiguring current containers too frequently.

Note that in both the training phase and the execution phase, the safety-aware action executor is commissioned to avoid the low-level scaling decisions that deteriorate the application performance.

5.5 Performance Evaluation

In this section, we evaluate the effectiveness of *DeepScale* using the real-world datasets. By implementing a prototype system in realistic multi-cloud, we compare the performance of *DeepScale* with state-of-the-art baselines. The highlights are:

- For different applications, *DeepScale* achieves up to 23% savings in terms of the cumulative deployment cost.

- In the meantime, *DeepScale* can achieve 100 percent satisfaction on the constraint of average response time.

5.5.1 Datasets

We collect the real container pricing schemes in April 2021 from three leading cloud providers, i.e., Amazon Elastic Container Service (ECS)[3], Microsoft Azure Container Instances[4], and Google Kubernetes Engine (GKE)[5]. 18 major Amazon, Azure, and Google data centers have been included in the experiments. Furthermore, we adopt 82 user regions from 35 countries on 6 continents in the Sprint IP Network[6] to simulate the global user community.

To evaluate the network latency between users and deployed services, we use the network latency information in the Sprint[7] IP backbone network databases.

We use real traces of user requests based on the public benchmark WikiBench [184] to create workloads for our prototype system. WikiBench[8] is a Web hosting benchmark allowing the stress-test of systems designed to host Web applications. Following [179], our workload contains 1% of all user requests issued to Wikipedia (in all languages). Referring to [198], we apply Facebook subscribers statistics[9] to simulate the distribution of application requests from different user regions.

5.5.2 Algorithm Implementation

We implement *DeepScale* using PyTorch [130] on a server with Intel Core i7-8700 CPU (3.2 GHz and 16 GB of RAM). The built DQN has two fully-connected hidden layers, each with 64 nodes. The input and hidden layers use Rectified Linear Units (ReLUs). We apply Adam [90] as the optimizer. The Adam optimizer is an effective method for stochastic optimiza-

[3]https://aws.amazon.com/fargate/pricing/
[4]https://azure.microsoft.com/en-us/pricing/details/container-instances/
[5]https://cloud.google.com/kubernetes-engine/pricing
[6]https://www.sprint.net/network_maps.php
[7]https://www.sprint.net/tools/ip-network-performance
[8]http://www.wikibench.eu/
[9]https://www.internetworldstats.com/facebook.htm

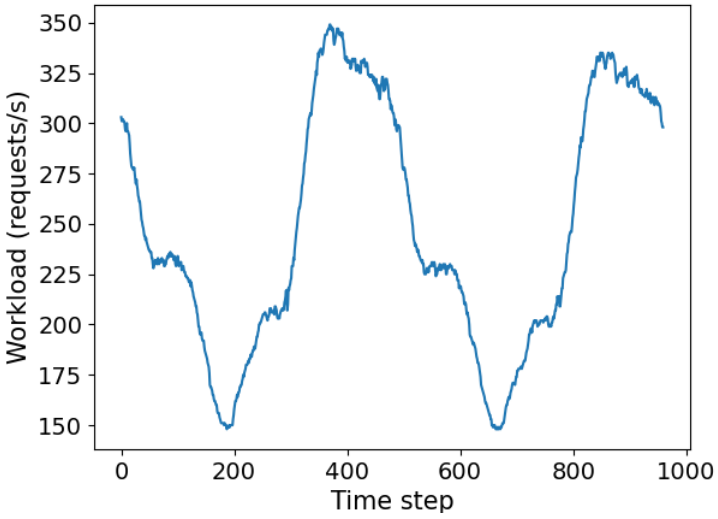


Figure 5.3: Request rate from WikiBench.

tion. The initial and minimum ϵ, i.e., the probability that DRL randomly chooses an action (in step 4 of Algorithm 11), are set as 0.2 and 0.01, respectively [73]. Other algorithm settings of DQN include: learning rate α is 0.001, discount factor γ is 1.0. The size of the experience replay buffer is 500 and the mini-batch size is 32. The DQN is trained for 200 episodes because it always converges within 200 episodes. Refer to [201], the acceptable threshold of ART, i.e., m in eq. (5.6), is set to 150 ms.

We randomly extract one day's workload in September 2007 from WikiBench[10] for training and use the workload on the following day for testing. The duration of each time period, i.e., the time interval between making scaling decisions, is set to 3 minutes as in [140], [141]. Figure 5.3 depicts the request rate (number of requests per second) during the two days (total 960 time periods). In our experiments, we apply 3 applications reported in [75]. The application processing time for a single request is approximate 10ms (*app-1*), 15ms (*app-2*), and 20ms (*app-3*) respectively running on

[10]http://www.wikibench.eu/wiki/2007-09/

the container with one vCPU unit. Each experiment is repeated independently 30 times.

5.5.3 Baselines

To evaluate the performance of *DeepScale*, we further implement an industry scaling strategy and two recently proposed baselines in our experiments.

Amazon auto-scaling service [13] provides many methods to control the number of replicas to be created to meet the increasing or decreasing workload of an application. We apply the rule-based auto-scaling method by setting an upper threshold (0.8) and a lower threshold (0.6) on the CPU utilization of containers [126]. For convenience, we denote the baseline algorithm as *AWS-Scale*. Concretely, at the end of each time interval, i.e., 3 minutes, if the average CPU utilization is above the upper threshold, *AWS-Scale* will scale-up the system. In case the CPU utilization is below the lower threshold, *AWS-Scale* will scale-down the system. For a fair comparison, the heuristics proposed in our action executor are used in *AWS-Scale* to make the scale-up and scale-down decisions in multi-cloud.

A-SARSA [213] is a recently proposed container auto-scaling algorithm based on reinforcement learning. *A-SARSA* first combines an ARIMA and a feedforward neural network to predict the CPU utilization and response time. Then the two predicted values are discretized into different levels respectively. Finally, a Q table is trained by SARSA to make scaling decisions. To avoid the response time beyond the predefined constraint, *A-SARSA* also applies a penalty-based reward function. Because *A-SARSA* only considers the horizontal scaling of containers in a single cloud data center, we also include the heuristic proposed in our action executor in *A-SARSA* for a fair comparison.

Deep Q-Learning Container Migration algorithm (*DQLCM*) [173] is proposed for delay-sensitive applications in fog computing. To select an ap-

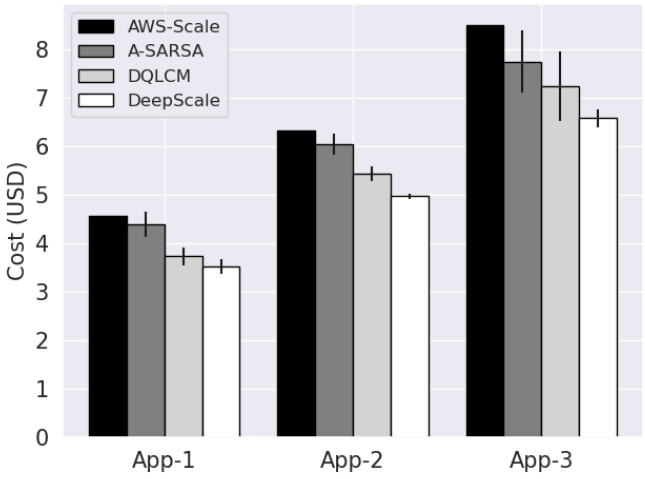


Figure 5.4: Cumulative deployment cost for different applications.

propriate action, *DQLCM* applies problem-specific strategies for container migration. Particularly, two thresholds of CPU utilization, i.e., th_{under} and th_{over}, are predefined to classify fog nodes into different groups, i.e., under-utilized nodes and over-utilized nodes. For the nodes in different groups, different heuristics are proposed to determine the migrated containers and their destination. The action set of *DQLCM* is defined as optional container placement generated by these heuristics. To adapt *DQLCM* to our problem, we regard container migration as container scaling and fog nodes as application replicas. As recommended in [173], th_{under} and th_{over} are set as 0.5 and 0.9 respectively to minimize the deployment cost subject to the constraint on the average response time.

5.5.4 Cost Comparison

Figure 5.4 demonstrates the cumulative deployment cost over the testing day. *DeepScale* saves cost by 23% for *app-1*, 21% for *app-2*, and 23% for *app-3* compared to *AWS-Scale*. We cannot gain significant improvements on the effectiveness of *AWS-Scale* by tuning the two thresholds of CPU utiliza-

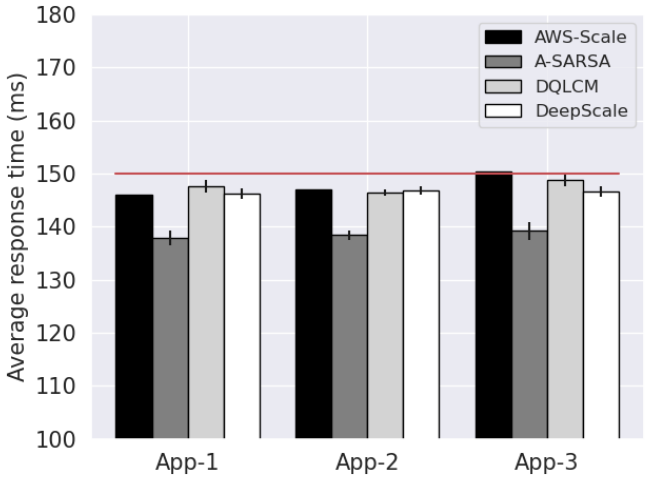


Figure 5.5: Average response time for different applications.

tion. That is, the cumulative deployment cost cannot be further reduced without violating the constraint on the average response time. This shows that the threshold-based method in the industry may be not suitable for scaling cloud applications with dynamically changing and widely distributed workloads. The bad performance on the deployment cost in our experiments is consistent with previous observations reported in [126].

A-SARSA spends 21% more deployment cost than *DeepScale* on average for different applications. Compared with the discretization technique adopted in *A-SARSA*, *DeepScale* exploits DQN to learn very complex functions and can handle high-dimension state space more effectively.

DeepScale also achieves average 8% less cost than *DQLCM*, because the scaling policies devised by *DQLCM* are less adaptive without a workload prediction model. The observed performance differences between *DeepScale* and all baselines are all verified through a statistical test (Wilcoxon Rank-Sum test) with a significance level of 0.05.

5.5.5 Constraint Compliance

Figure 5.5 shows the average response time over the testing day achieved by *DeepScale* and the baselines. The red line in Figure 5.5 is the predefined constraint, i.e., m. As shown in Figure 5.5, for *app-3*, the average response time of *AWS-Scale* is slightly over m. We have tried to reduce the upper threshold of *AWS-Scale*, e.g., from 0.8 to 0.75, to meet the constraint. However, the new threshold significantly increases the deployment cost. Among the three RL-based approaches, *A-SARSA* always has the lowest average response time. For *DQLCM*, the average response time is longer than *DeepScale* for *app-1* and *app-3*, while slightly shorter than *DeepScale* for *app-2*.

From the above results, we can conclude that our proposed *DeepScale* can obtain the lowest cumulative deployment cost and also guarantee constraint satisfaction. Furthermore, from Figure 5.4 we observe that the cumulative deployment cost has small standard deviations over 30 repeated experiments, confirming its stability and reliability for containerized application scaling in multi-cloud.

5.5.6 Analysis

First, we evaluate the accuracy of our LSTM-based workload predictor in terms of Root-Mean-Square Error (RMSE). The values of RMSE for the training workload and the testing workload are 1.8 and 1.63 requests/s respectively. The high test accuracy demonstrates the effectiveness of our LSTM-based workload predictor.

Next, we depict the change of the cumulative deployment cost and average response time obtained by *DeepScale* on the testing day across all learning episodes in Figure 5.6 and Figure 5.7. For the ablation study, the results of *DeepScale* without the LSTM-based prediction model are also included in Figure 5.6 and Figure 5.7. We only present the results for *app-3* because we can observe a similar trend for other applications. Figure 5.6

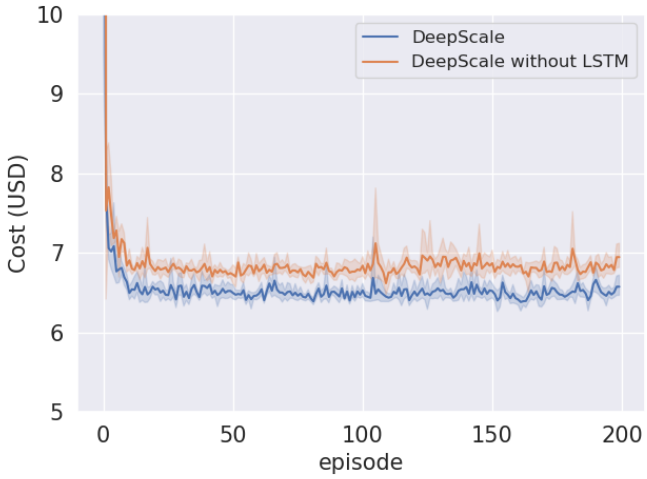


Figure 5.6: Cumulative cost for *app-3* in one episode.

shows that the cumulative deployment cost becomes flattened after about 50 episodes. With the LSTM model, *DeepScale* saves the deployment cost by 7% without increasing the average response time. By considering the future workloads, *DeepScale* is more effective when making scaling decisions. In Figure 5.7, we can observe that the average response time falls strictly under m (red line) after about 100 episodes (*DeepScale*) and 150 episodes (*DeepScale* without LSTM model), which shows that using LSTM is also helpful to reduce the time required for *DeepScale* to learn constraint-compliant policies.

After training, all the three RL-based approaches can scale containerized applications for incoming requests with trivial computational overhead. The total time required to make a high-level scaling decision using the DQN and a low-level scaling decision through a safety-aware action executor is within 1 ms. The training time of *DeepScale* is within 30 minutes, which includes the training of the LSTM-based workload predictor and the DRL-based scaling policy. Periodical use of *DeepScale* every day is highly feasible in practice. The training time of *A-SARSA* is similar to *DeepScale*. The training of *DQLCM* takes a much longer time due to a more

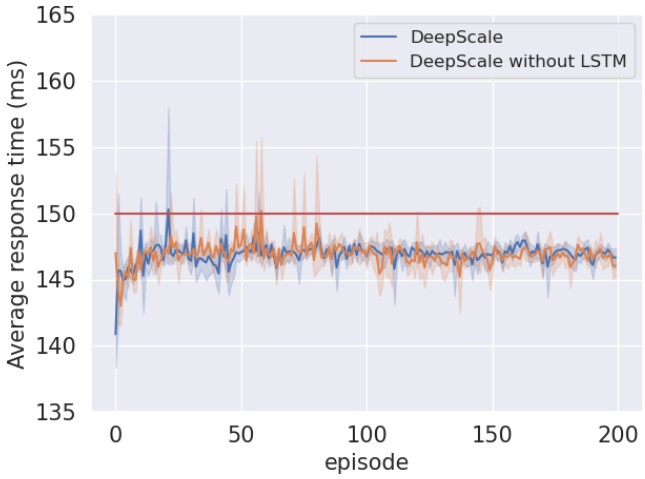


Figure 5.7: Average response time for *app-3* in one episode.

complex action space (about 5 hours).

5.6 Chapter Summary

In this chapter, we first formulate the *EAD* problem to minimize the cumulative deployment cost over a time span under the constraint on the average response time. Secondly, we propose a novel approach, namely *DeepScale*, combing DRL with an LSTM prediction model, to solve the *EAD* problem automatically. With newly designed safe mechanisms, *DeepScale* can ensure that any scaling decisions made by DRL will not deteriorate the application performance. Finally, we implement a fully functioning prototype of *DeepScale* using PyTorch and conduct extensive experiments on real-world datasets. The experiments with realistic Web application workloads show that *DeepScale* can significantly reduce the deployment cost of applications compared with the state-of-the-art baselines, including Amazon auto-scaling service and recently proposed RL-based approaches. In the meanwhile, *DeepScale* can effectively satisfy the constraint on the aver-

age response time for different applications.

Chapter 6

Conclusions and Future Works

The overall goal of this thesis is to solve location-aware application deployment in multi-cloud. This research goal has been successfully fulfilled by proposing innovative optimization methods and machine learning techniques for three major scenarios, namely Composite Application Deployment (*CAD*), Application Replication and Deployment (*ARD*), and Elastic Application Deployment (*EAD*). For each scenario, we first formulated the problem to capture the key characteristics of the studied scenario. Then, we developed new effective approaches based on the characteristics of these scenarios. Specifically, we investigated and proposed different optimization and machine learning frameworks for solving these problems. These proposed approaches were compared with state-of-the-art approaches using real-world datasets. Finally, we provided insights regarding the practical value of the research work in the industry.

The remainder of this chapter is organized as follows. Section 6.1 outlines the objectives that have been achieved in this thesis. Section 6.2 presents the main conclusions reached in this work. Section 6.3 explores possible future work directions.

6.1 Achieved Objectives

This thesis contributes to the fields of multi-cloud application deployment and relevant algorithm designs. The major contributions are listed as follows.

1. This research starts with the location-aware *CAD* problem that has not been properly studied in the literature. Chapter 3 first defines the objective, variables, and constraints of the service deployment problem for composite applications in multi-cloud. Then, this chapter proposes a hybrid GA-based approach, i.e., *H-GA*, for solving the *CAD* problem. *H-GA* features a newly designed and domain-tailored service clustering algorithm, repair algorithm, solution representation, population initialization, and genetic operators. Specifically, the proposed service clustering algorithm clusters dependent constituent services and deploys the services in the same cluster to the same data center. The clustering algorithm can significantly reduce the size of the search space of the *CAD* problem. By transforming a self-adaptive subset of over-budget solutions into budget feasible solutions, the proposed repair algorithm can improve solution quality and reduce the computation time of *H-GA*. Experiments show that *H-GA* significantly outperforms the existing approaches, achieving up to about 8% performance improvement in terms of response time, and 100% budget satisfaction in the meantime.

2. To satisfy the requirements of some cloud applications on low average response time, this thesis proposes two approaches under different optimization frameworks to solve the location-aware *ARD* problem (Chapter 4). For the *ARD with close dispatching* problem, we developed an approach under the GA framework, i.e., *GA-ARD*. *GA-ARD* features problem-specific solution representation, fitness measurement, and population initialization, which can effectively optimize the deployment of application replicas in multi-cloud. The ex-

periments show that *GA-ARD* outperforms industry-leading application replication and placement strategies. For the *ARD with flexible dispatching* problem, we develop another approach under a two-stage optimization framework, i.e., *MCApp*. *MCApp* can optimize both replica deployment and request dispatching by combining an iterative MILP-based algorithm and a domain-tailored LNS-based algorithm. Our experiments show that *MCApp* can achieve up to 25% reduction in total deployment cost compared with several recently developed approaches.

3. This thesis proposes a machine learning approach for the *EAD* problem. To the best of our knowledge, this is the first study in the literature on automatically scaling containerized applications in multi-cloud (Chapter 5). The proposed DRL-based algorithm, i.e., *DeepScale*, applies a DQN to capture the optimal scaling policy that can perform online adaptive scaling. Particularly, the scaling policy is trained to minimize total deployment cost over a time span while satisfying the constraint on average response time. The trained policy facilitates computationally efficient container scaling in multi-cloud. *DeepScale* includes an LSTM-based workload predictor to allow the DQN to consider predicted future requests while making cost-effective scaling decisions. Besides, we design a penalty-based reward function and safety-aware action executor to ensure that any scaling decisions made by DRL will satisfy the performance constraint. The experiments show that *DeepScale* can significantly reduce the deployment cost of applications compared with the state-of-the-art baselines, including Amazon auto-scaling service and recently proposed RL-based algorithms. In the meanwhile, the novel safe mechanisms within *DeepScale* can effectively satisfy the constraint on the average response time for different applications.

6.2 Conclusions

This section outlines the main conclusions reached in the major contribution chapters presented in this thesis (Chapter 3 to Chapter 5). These conclusions are summarized in Table 6.1.

Firstly, three problem formulations were introduced from Chapter 3 to Chapter 5. The formulated *CAD*, *ARD*, and *EAD* problems have different optimization decisions to satisfy the practical requirements of multi-cloud application deployment. The objectives and constraints of the three problems focus on the application deployment cost and average response time. Secondly, two approaches under the GA framework, i.e., *H-GA* and *GA-ARD*, were developed to address the *CAD* problem and the *ARD with close dispatching* problem respectively. A problem-specific service clustering algorithm, population initialization, and fitness evaluation were designed for effective application deployment. Thirdly, a two-stage optimization approach, namely *MCApp*, was proposed to optimize both the deployment of application replicas and dispatching of user requests for the *ARD with flexible dispatching* problem. Lastly, a machine learning approach, i.e., *DeepScale*, was proposed for the *EAD* problem. Moreover, different constraint handling techniques were applied to satisfy the requirements of budgetary control or application performance, e.g., repair algorithms for *H-GA* and safe exploration for *DeepScale*.

6.2.1 Problem Formulation

This thesis introduced three novel problem formulations under two public cloud paradigms, i.e., IaaS and CaaS, for multi-cloud application deployment considering the key impact of the location on both the deployment cost and the application response time. For two request dispatching mechanisms for *ARD*, i.e., *ARD with close dispatching* and *ARD with flexible dispatching*, four types of optimization decisions were identified. The locations of VMs and containers were considered in all formulated prob-

Table 6.1: Summary conclusions to address the research goal in the thesis.

	CAP	ARD		*EAD*
Paradigm	Multi-cloud IaaS			Multi-cloud CaaS
Optimization decisions	Deployment of constituent services (without replication)	Deployment of application replicas (close dispatching)	Deployment of application replicas and dispatching of user requests	Elastic deployment of application replicas
Objectives	Minimizing average response time	Minimizing total deployment cost		Minimizing cumulative cost
Constraints	Budget	Performance threshold		
Proposed approach	*H-GA*	*GA-ARD*	*MCApp*	*DeepScale*
Optimization frameworks	GA		Two-stage optimization	DRL
Featuring techniques	Service clustering, repair algorithm and population initialization	Fitness measurement and population initialization	Iterative MILP and domain-tailored LNS	LSTM-based prediction model and safe RL mechanisms

lems because they significantly affect the deployment cost and average response time of applications. The cost and performance were considered as the objectives and constraints of the three problems. Afterward, they were used to evaluate the proposed approaches.

6.2.2 GA Optimization Framework

This thesis proposed two novel approaches under the GA optimization framework, i.e., *H-GA* for the *CAD* problem and *GA-ARD* for the *ARD with close dispatching* problem. For the chromosome representation, *H-GA* applies two-string decoding for both the VM location and VM type. Particularly, by clustering dependent services, the services in the same cluster are treated as a single deployment unit in any VM location string, which significantly reduces the search space. The chromosome representation of *GA-ARD* is only for the VM location because the placement of VMs where application replicas are deployed directly affects the dispatching of user requests for the *ARD with close dispatching* problem. Both *H-GA* and *GA-ARD* apply the heuristic-based methods to generate the initial population. The seeding strategy improves the solution quality and convergence speed. Other algorithmic novelties of the two GA-based approaches include problem-specific genetic operators (*H-GA*) and fitness measurement (*GA-ARD*).

6.2.3 Two-stage Optimization Framework

The *ARD with flexible dispatching* problem simultaneously optimizes the deployment of application replicas and the dispatching of user requests. The complexity of optimization decisions motivated us to adopt a two-stage optimization framework to progressively improve the solution quality. Concretely, our proposed approach, namely *MCApp*, first transforms the problem into a series of MILP problems through bounding the utilization rate of VMs for application replicas. A MILP-based algorithm was

proposed to effectively generate a base solution with good resource utilization. To explore the search space, a problem-specific LNS-based algorithm was developed to further optimize the base solution. The LNS-based algorithm includes the new destroy heuristic and repair heuristic to improve the deployment of application replicas and a delay-oriented heuristic to dispatch user requests in order to achieve high performance based on the current replica deployment.

6.2.4 Machine Learning Techniques

In this thesis, machine learning techniques were developed to solve the *EAD* problem. On the one hand, we sought to utilize an LSTM neural network to predict the workload of cloud applications. On the other hand, we adopted DRL to devise effective scaling policies for applications with dynamic and widely distributed workloads. Our proposed algorithm, i.e., *DeepScale*, can consider the predicted future workload and perform both vertical scaling and horizontal scaling for containers in multi-cloud through the learned scaling policies.

6.2.5 Constraint Handling Methods

The *CAD*, *ARD*, and *EAD* problems are all constrained problems. Under different optimization frameworks and machine learning techniques, we proposed different methods for handling constraints in this thesis. For *H-GA*, we developed a repair algorithm during the evolution process. The repair algorithm can transform a self-adaptive subset of over-budget solutions into budget feasible solutions by iteratively reducing the total deployment cost to be within the given budget. The novel repair algorithm can achieve a desirable trade-off between performance and computation. The constraint handling method of *GA-ARD* is integrated into the fitness measurement of GA. That is, a GAIN-based heuristic was designed to ensure the average response time is below the acceptable threshold. For

MCApp, the solutions generated by the LNS-based algorithm are constraint-compliant through the domain-tailored repair heuristic. Finally, *DeepScale* innovatively applies two safety mechanisms, i.e., a penalty-based reward function and a safety-aware action executor, to ensure that any scaling decisions made by DRL will satisfy the performance constraint.

6.3 Future Work

Due to the scope of this research, there are still some areas for potential extensions and future work. This section briefly gives some research directions related to deployment scenarios and algorithmic techniques.

6.3.1 Deployment Scenarios

For location-aware application deployment, data sovereignty [80] and edge computing [160] are two promising directions.

Data Sovereignty

Multi-cloud deployment imposes inherent security and privacy concerns regarding data analysis and exchange for some applications that handle sensitive information [44]. For example, citizen data of a smart city application being stored in cloud may be relocated to a data center in a different jurisdiction or accessible to users located in different jurisdictions. To address the data sovereignty issues, location-aware data placement [128] and location-based data encryption [44] can be applied. Note that both the two solutions impact the deployment cost and performance of applications, which deserve to be studied in future work.

Edge Computing

In practice, the multi-cloud solutions can integrate infrastructures and operations across private clouds, public clouds, and edge [188]. For some latency-sensitive applications, edge computing is an appropriate solution [214]. In edge computing, a significant volume of light computation and storage infrastructures, called edge servers, are deployed close to users [202]. In this case, application requests can be offloaded to suitable edge servers to shorten response time. Currently, leading cloud providers have started to deliver the edge services to locations close to the large population and industrial centers, e.g., AWS Outposts[1] and Local Zones[2]. The deployment cost of applications in edge is based on a different pricing scheme[3] from cloud. The deployment solutions combining cloud with edge is still an open question to researchers. Future work can be explored to consider the extra resource layer provided by edge computing.

6.3.2 Algorithmic Techniques

For algorithmic design, the following techniques are promising to satisfy some practical requirements of multi-cloud application deployment.

Multi-objective Approaches

In most cases, application providers have exact requirements on the budget and performance of application deployment. However, sometimes it is difficult for application providers to specify budget information or acceptable performance threshold. Besides, a high level of domain expertise is required to exactly weigh any conflicting objectives, such as total deployment cost and average response time. Multi-Objective Evolutionary Algorithms (MOEAs) can be utilized to search for the Pareto front, in which

[1]https://aws.amazon.com/outposts/
[2]https://aws.amazon.com/about-aws/global-infrastructure/localzones/
[3]https://aws.amazon.com/outposts/pricing/

each solution represents a unique trade-off deployment solution in consideration of all conflicting objectives. To efficiently find non-dominated deployment solutions, new clustering methods can be designed to group application users according to their locations. The cluster analysis information will help to expedite the search for good solutions.

Multi-agent RL

In this thesis, we apply the centralized application deployment for the *EAD* problem. The centralized approaches may suffer from the lack of scalability, especially when considering a larger number of data centers and user regions. It is necessary to investigate the decentralized application deployment approaches to overcome the scalability problem. Novel multi-agent RL-based approaches can be developed for the decentralized *EAD* problem with multiple cooperative deployment centers in multi-cloud. Efficient agent coordination mechanisms are needed to realize multi-agent coordination.

Bibliography

[1] AARTS, E., AARTS, E. H., AND LENSTRA, J. K. *Local search in combinatorial optimization*. Princeton University Press, 2003.

[2] ABBEEL, P., COATES, A., AND NG, A. Y. Autonomous helicopter aerobatics through apprenticeship learning. *The International Journal of Robotics Research 29*, 13 (2010), 1608–1639.

[3] ACHIAM, J., HELD, D., TAMAR, A., AND ABBEEL, P. Constrained policy optimization. In *International Conference on Machine Learning* (2017), PMLR, pp. 22–31.

[4] AHUJA, R. K., ERGUN, Ö., ORLIN, J. B., AND PUNNEN, A. P. A survey of very large-scale neighborhood search techniques. *Discrete Applied Mathematics 123*, 1-3 (2002), 75–102.

[5] AL-DHURAIBI, Y., PARAISO, F., DJARALLAH, N., AND MERLE, P. Autonomic vertical elasticity of docker containers with elasticdocker. In *2017 IEEE 10th international conference on cloud computing (CLOUD)* (2017), IEEE, pp. 472–479.

[6] ALDWYAN, Y., SINNOTT, R. O., AND JAYAPUTERA, G. T. Elastic deployment of container clusters across geographically distributed cloud data centers for web applications. *Concurrency and Computation: Practice and Experience*, e6436.

[7] ALLEN, A. O. *Probability, statistics, and queueing theory*. Academic press, 2014.

[8] AMODEI, D., OLAH, C., STEINHARDT, J., CHRISTIANO, P., SCHULMAN, J., AND MANÉ, D. Concrete problems in ai safety. *arXiv preprint arXiv:1606.06565* (2016).

[9] ARABNEJAD, H., AND BARBOSA, J. G. A budget constrained scheduling algorithm for workflow applications. *Journal of grid computing 12*, 4 (2014), 665–679.

[10] ARABNEJAD, H., PAHL, C., JAMSHIDI, P., AND ESTRADA, G. A comparison of reinforcement learning techniques for fuzzy cloud auto-scaling. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)* (2017), IEEE, pp. 64–73.

[11] ARULKUMARAN, K., DEISENROTH, M. P., BRUNDAGE, M., AND BHARATH, A. A. A brief survey of deep reinforcement learning. *arXiv preprint arXiv:1708.05866* (2017).

[12] ARULKUMARAN, K., DEISENROTH, M. P., BRUNDAGE, M., AND BHARATH, A. A. Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine 34*, 6 (2017), 26–38.

[13] AWS. Amazon web services. https://aws.amazon.com/, 2020.

[14] BACK, T., AND SCHWEFEL, H.-P. Evolutionary computation: An overview. In *Proceedings of IEEE International Conference on Evolutionary Computation* (1996), IEEE, pp. 20–29.

[15] BAI, W., CHEN, L., CHEN, K., HAN, D., TIAN, C., AND WANG, H. Pias: Practical information-agnostic flow scheduling for commodity data centers. *IEEE/ACM Transactions on Networking (TON) 25*, 4 (2017), 1954–1967.

[16] BARKER, A., AND VAN HEMERT, J. Scientific workflow: a survey and research directions. In *International Conference on Parallel Processing and Applied Mathematics* (2007), Springer, pp. 746–753.

[17] BENIFA, J. B., AND DEJEY, D. Rlpas: Reinforcement learning-based proactive auto-scaler for resource provisioning in cloud environment. *Mobile Networks and Applications 24*, 4 (2019), 1348–1363.

[18] BENJAPONPITAK, T., KARAKATE, M., AND SRIPANIDKULCHAI, K. Enabling live migration of containerized applications across clouds. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications* (2020), IEEE, pp. 2529–2538.

[19] BJÖRKQVIST, M., CHEN, L. Y., AND BINDER, W. Dynamic replication in service-oriented systems. In *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)* (2012), IEEE, pp. 531–538.

[20] BJÖRKQVIST, M., CHEN, L. Y., AND BINDER, W. Opportunistic service provisioning in the cloud. In *2012 IEEE Fifth International Conference on Cloud Computing* (2012), IEEE, pp. 237–244.

[21] BONVIN, N., PAPAIOANNOU, T. G., AND ABERER, K. Autonomic sla-driven provisioning for cloud applications. In *2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing* (2011), IEEE, pp. 434–443.

[22] BUYYA, R., RANJAN, R., AND CALHEIROS, R. N. Intercloud: Utility-oriented federation of cloud computing environments for scaling of application services. In *International Conference on Algorithms and Architectures for Parallel Processing* (2010), Springer, pp. 13–31.

[23] CALHEIROS, R. N., MASOUMI, E., RANJAN, R., AND BUYYA, R. Workload prediction using ARIMA model and its impact on cloud

applications' QoS. *IEEE Transactions on Cloud Computing 3*, 4 (2014), 449–458.

[24] CALHEIROS, R. N., RANJAN, R., AND BUYYA, R. Virtual machine provisioning based on analytical performance and QoS in cloud computing environments. In *2011 International Conference on Parallel Processing* (2011), IEEE, pp. 295–304.

[25] CARTER, S., MACDONALD, N. J., AND CHENG, D. C. *Basic finance for marketers*, vol. 1. Food & Agriculture Org., 1997.

[26] CHEN, T., BAHSOON, R., AND TAWIL, A.-R. H. Scalable service-oriented replication with flexible consistency guarantee in the cloud. *Information Sciences 264* (2014), 349–370.

[27] CHEN, T., LI, M., AND YAO, X. On the effects of seeding strategies: a case for search-based multi-objective service composition. In *Proceedings of the genetic and evolutionary computation conference* (2018), pp. 1419–1426.

[28] CHEN, Y., AND TSAI, W.-T. *Service-oriented computing and web software integration: from principles to development*. Kendall/Hunt Publishing Co., 2012.

[29] CHENG, M., LI, J., AND NAZARIAN, S. Drl-cloud: Deep reinforcement learning-based resource provisioning and task scheduling for cloud service providers. In *Proceedings of the 23rd Asia and South Pacific Design Automation Conference* (2018), IEEE Press, pp. 129–134.

[30] CHENG, R., OROSZ, G., MURRAY, R. M., AND BURDICK, J. W. End-to-end safe reinforcement learning through barrier functions for safety-critical continuous control tasks. In *Proceedings of the AAAI Conference on Artificial Intelligence* (2019), vol. 33, pp. 3387–3395.

[31] CHINCHALI, S., HU, P., CHU, T., SHARMA, M., BANSAL, M., MISRA, R., PAVONE, M., AND KATTI, S. Cellular network traffic scheduling with deep reinforcement learning. In *Thirty-Second AAAI Conference on Artificial Intelligence* (2018).

[32] CHOOTINAN, P., AND CHEN, A. Constraint handling in genetic algorithms using a gradient-based repair method. *Computers & operations research 33*, 8 (2006), 2263–2281.

[33] COELLO, C. A. C. Use of a self-adaptive penalty approach for engineering optimization problems. *Computers in Industry 41*, 2 (2000), 113–127.

[34] COELLO, C. A. C. Constraint-handling techniques used with evolutionary algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion* (2021), pp. 692–714.

[35] COELLO, C. A. C., LAMONT, G. B., VAN VELDHUIZEN, D. A., ET AL. *Evolutionary algorithms for solving multi-objective problems*, vol. 5. Springer, 2007.

[36] COELLO COELLO, C. A. Constraint-handling techniques used with evolutionary algorithms. In *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion* (2016), ACM, pp. 563–587.

[37] COLUMBUS, L. 83% of enterprise workloads will be in the cloud by 2020. https://www.forbes.com/sites/louiscolumbus/2018/01/07/83-of-enterprise-workloads-will-be-in-the-cloud-by-2020/, 2018.

[38] DA SILVA, A. S., MA, H., MEI, Y., AND ZHANG, M. A hybrid memetic approach for fully automated multi-objective web service composition. In *2018 IEEE International Conference on Web Services (ICWS)* (2018), IEEE, pp. 26–33.

[39] DAVIS, L. *Handbook of genetic algorithms*. CumInCAD, 1991.

[40] DE ALFONSO, C., CALATRAVA, A., AND MOLTÓ, G. Container-based virtual elastic clusters. *Journal of Systems and Software 127* (2017), 1–11.

[41] DRIESSENS, K., AND DŽEROSKI, S. Integrating guidance into relational reinforcement learning. *Machine Learning 57*, 3 (2004), 271–304.

[42] EDUCATION, I. C. Iaas vs. paas vs. saas. https://www.ibm.com/cloud/learn/iaas-paas-saas, 2021.

[43] ESBENSEN, H. Computing near-optimal solutions to the steiner problem in a graph using a genetic algorithm. *Networks 26*, 4 (1995), 173–185.

[44] ESPOSITO, C., CASTIGLIONE, A., FRATTINI, F., CINQUE, M., YANG, Y., AND CHOO, K.-K. R. On data sovereignty in cloud-based computation offloading for smart cities applications. *IEEE Internet of Things Journal 6*, 3 (2018), 4521–4535.

[45] FERRER, A. J., HERNÁNDEZ, F., TORDSSON, J., ELMROTH, E., ALI-ELDIN, A., ZSIGRI, C., SIRVENT, R., GUITART, J., BADIA, R. M., DJEMAME, K., ET AL. Optimis: A holistic approach to cloud service provisioning. *Future Generation Computer Systems 28*, 1 (2012), 66–77.

[46] FOX, A., GRIFFITH, R., JOSEPH, A., KATZ, R., KONWINSKI, A., LEE, G., PATTERSON, D., RABKIN, A., AND STOICA, I. Above the clouds: A berkeley view of cloud computing. *Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS 28*, 13 (2009), 2009.

[47] GANDHI, A., DUBE, P., KARVE, A., KOCHUT, A., AND ZHANG, L. Providing performance guarantees for cloud-deployed applications. *IEEE Transactions on Cloud Computing 8*, 1 (2017), 269–281.

[48] GAO, Y., GUAN, H., QI, Z., HOU, Y., AND LIU, L. A multi-objective ant colony system algorithm for virtual machine placement in cloud computing. *Journal of Computer and System Sciences 79*, 8 (2013), 1230–1242.

[49] GARCIA, J., AND FERNÁNDEZ, F. Safe exploration of state and action spaces in reinforcement learning. *Journal of Artificial Intelligence Research 45* (2012), 515–564.

[50] GARCIA, J., AND FERNÁNDEZ, F. A comprehensive survey on safe reinforcement learning. *Journal of Machine Learning Research 16*, 1 (2015), 1437–1480.

[51] GEHRING, C., AND PRECUP, D. Smart exploration in reinforcement learning using absolute temporal difference errors. In *Proceedings of the 2013 international conference on Autonomous agents and multi-agent systems* (2013), pp. 1037–1044.

[52] GEN, M., AND CHENG, R. A survey of penalty techniques in genetic algorithms. In *Proceedings of IEEE International Conference on Evolutionary Computation* (1996), IEEE, pp. 804–809.

[53] GEORGIOS, C., EVANGELIA, F., CHRISTOS, M., AND MARIA, N. Exploring cost-efficient bundling in a multi-cloud environment. *Simulation Modelling Practice and Theory 111* (2021), 102338.

[54] GHANBARI, H., LITOIU, M., PAWLUK, P., AND BARNA, C. Replica placement in cloud through simple stochastic model predictive control. In *2014 IEEE 7th International Conference on Cloud Computing* (2014), IEEE, pp. 80–87.

[55] GHANI, N., SHAMI, A., ASSI, C., AND RAJA, M. Intra-onu bandwidth scheduling in ethernet passive optical networks. *IEEE Communications Letters 8*, 11 (2004), 683–685.

[56] GLOVER, F., AND LAGUNA, M. Tabu search. In *Handbook of combinatorial optimization*. Springer, 1998, pp. 2093–2229.

[57] GOASDUFF, L. Why organizations choose a multicloud strategy. https://www.gartner.com/smarterwithgartner/why-organizations-choose-a-multicloud-strategy, 2019.

[58] GOOGLE. Ortools. https://developers.google.com/optimization.

[59] GROZEV, N., AND BUYYA, R. Inter-cloud architectures and application brokering: taxonomy and survey. *Software: Practice and Experience 44*, 3 (2014), 369–390.

[60] GUO, J., LIU, F., WANG, T., AND LUI, J. C. Pricing intra-datacenter networks with over-committed bandwidth guarantee. In *Proc. USENIX ATC* (2017).

[61] GUZEK, M., BOUVRY, P., AND TALBI, E.-G. A survey of evolutionary computation for resource management of processing in cloud computing. *IEEE Computational Intelligence Magazine 10*, 2 (2015), 53–67.

[62] HAGAN, M. T., DEMUTH, H. B., BEALE, M. H., AND DE JESÚS, O. *Neural network design*, vol. 20. Pws Pub. Boston, 1996.

[63] HALLAM, J. W., AKMAN, O., AND AKMAN, F. Genetic algorithms with shrinking population size. *Computational Statistics 25*, 4 (2010), 691–705.

[64] HAN, Z., TAN, H., CHEN, G., WANG, R., CHEN, Y., AND LAU, F. C. Dynamic virtual machine management via approximate markov decision process. In *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications* (2016), IEEE, pp. 1–9.

[65] HANSEN, P., MLADENOVIĆ, N., AND MORENO PEREZ, J. A. Variable neighbourhood search: methods and applications. *Annals of Operations Research 175*, 1 (2010), 367–407.

[66] HASSANZADEH-NAZARABADI, Y., KÜPÇÜ, A., AND OZKASAP, O. Decentralized utility-and locality-aware replication for heterogeneous DHT-based P2P cloud storage systems. *IEEE Transactions on Parallel and Distributed Systems 31*, 5 (2019), 1183–1193.

[67] HAT, R. Red hat openshift container platform. https://www.openshift.com/products/container-platform, 2021.

[68] HEILIG, L., BUYYA, R., AND VOSS, S. Location-aware brokering for consumers in multi-cloud computing environments. *Journal of Network and Computer Applications 95* (2017), 79–93.

[69] HEILIG, L., LALLA-RUIZ, E., AND VOSS, S. A cloud brokerage approach for solving the resource management problem in multi-cloud environments. *Computers & Industrial Engineering 95* (2016), 16–26.

[70] HOCHREITER, S., AND SCHMIDHUBER, J. Long short-term memory. *Neural computation 9*, 8 (1997), 1735–1780.

[71] HOLLAND, J. H. Outline for a logical theory of adaptive systems. *Journal of the ACM (JACM) 9*, 3 (1962), 297–314.

[72] HOROVITZ, S., AND ARIAN, Y. Efficient cloud auto-scaling with sla objective using q-learning. In *2018 IEEE 6th International Conference on Future Internet of Things and Cloud (FiCloud)* (2018), IEEE, pp. 85–92.

[73] HU, J., GU, J., SUN, G., AND ZHAO, T. A scheduling strategy on load balancing of virtual machine resources in cloud computing environment. In *Parallel Architectures, Algorithms and Programming*

(PAAP), 2010 Third International Symposium on (2010), IEEE, pp. 89–96.

[74] HUANG, H., MA, H., AND ZHANG, M. An enhanced genetic algorithm for web service location-allocation. In *International Conference on Database and Expert Systems Applications* (2014), Springer, pp. 223–230.

[75] HUANG, K.-C., AND SHEN, B.-J. Service deployment strategies for efficient execution of composite saas applications on cloud platform. *Journal of Systems and Software 107* (2015), 127–141.

[76] HUANG, V., CHEN, G., ZHANG, P., LI, H., HU, C., PAN, T., AND FU, Q. A scalable approach to sdn control plane management: High utilization comes with low latency. *IEEE Transactions on Network and Service Management 17*, 2 (2020), 682–695.

[77] HÖLZLE, U. How computing has evolved, and why you need a multi-cloud strategy. https://cloud.google.com/blog/topics/hybrid-cloud/future-isnt-just-cloud-its-multi-cloud, 2020.

[78] IBM. What is containers as a service (caas)? https://www.ibm.com/services/cloud/containers-as-a-service, 2021.

[79] IMDOUKH, M., AHMAD, I., AND ALFAILAKAWI, M. G. Machine learning-based auto-scaling for containerized applications. *Neural Computing and Applications 32*, 13 (2020), 9745–9760.

[80] IRION, K. Government cloud computing and national data sovereignty. *Policy & Internet 4*, 3-4 (2012), 40–71.

[81] ISENBERG, K. IaaS vs. CaaS vs. PaaS vs. FaaS: Choosing the right platform. https://dzone.com/articles/iaas-vs-caas-vs-paas-vs-faas-choosing-the-right-platform, 2017.

[82] JAMSHIDI, P., SHARIFLOO, A., PAHL, C., ARABNEJAD, H., METZGER, A., AND ESTRADA, G. Fuzzy self-learning controllers for elasticity management in dynamic cloud architectures. In *2016 12th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA)* (2016), IEEE, pp. 70–79.

[83] JENNINGS, B., AND STADLER, R. Resource management in clouds: Survey and research challenges. *Journal of Network and Systems Management 23*, 3 (2015), 567–619.

[84] JOHNSON, D. S., PAPADIMITRIOU, C. H., AND YANNAKAKIS, M. How easy is local search? *Journal of computer and system sciences 37*, 1 (1988), 79–100.

[85] JRAD, F., TAO, J., BRANDIC, I., AND STREIT, A. SLA enactment for large-scale healthcare workflows on multi-cloud. *Future Generation Computer Systems 43* (2015), 135–148.

[86] KARDANI-MOGHADDAM, S., BUYYA, R., AND RAMAMOHANARAO, K. Adrl: A hybrid anomaly-aware deep reinforcement learning-based resource scaling in clouds. *IEEE Transactions on Parallel and Distributed Systems 32*, 3 (2020), 514–526.

[87] KAUR, A., GUPTA, P., SINGH, M., AND NAYYAR, A. Data placement in era of cloud computing: a survey, taxonomy and open research issues. *Scalable Computing: Practice and Experience 20*, 2 (2019), 377–398.

[88] KHALAJZADEH, H., YUAN, D., GRUNDY, J., AND YANG, Y. Improving cloud-based online social network data placement and replication. In *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)* (2016), IEEE, pp. 678–685.

[89] KHAZAEI, H., MISIC, J., AND MISIC, V. B. Performance analysis of cloud computing centers using m/g/m/m+ r queuing systems.

IEEE Transactions on parallel and distributed systems 23, 5 (2011), 936–943.

[90] KINGMA, D. P., AND BA, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[91] KUMAR, J., GOOMER, R., AND SINGH, A. K. Long short term memory recurrent neural network (lstm-rnn) based workload forecasting model for cloud datacenters. *Procedia Computer Science 125* (2018), 676–682.

[92] L., L. Cron job: A comprehensive guide for beginners 2021. https://www.hostinger.com/tutorials/cron-job, 2020.

[93] LEVINE, S., FINN, C., DARRELL, T., AND ABBEEL, P. End-to-end training of deep visuomotor policies. *The Journal of Machine Learning Research 17*, 1 (2016), 1334–1373.

[94] LI, W., YANG, Y., AND YUAN, D. A novel cost-effective dynamic data replication strategy for reliability in cloud data centres. In *2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing* (2011), IEEE, pp. 496–502.

[95] LILLICRAP, T. P., HUNT, J. J., PRITZEL, A., HEESS, N., EREZ, T., TASSA, Y., SILVER, D., AND WIERSTRA, D. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971* (2015).

[96] LIN, L.-J. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning 8*, 3-4 (1992), 293–321.

[97] LIPOWSKI, A., AND LIPOWSKA, D. Roulette-wheel selection via stochastic acceptance. *Physica A: Statistical Mechanics and its Applications 391*, 6 (2012), 2193–2196.

[98] LITTLE, J. D., AND GRAVES, S. C. Little's law. In *Building intuition*. Springer, 2008, pp. 81–100.

[99] LIU, L., ZHANG, M., BUYYA, R., AND FAN, Q. Deadline-constrained coevolutionary genetic algorithm for scientific workflow scheduling in cloud computing. *Concurrency and Computation: Practice and Experience 29*, 5 (2017), e3942.

[100] LIU, N., LI, Z., XU, J., XU, Z., LIN, S., QIU, Q., TANG, J., AND WANG, Y. A hierarchical framework of cloud resource allocation and power management using deep reinforcement learning. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)* (2017), IEEE, pp. 372–382.

[101] LIU, X., LAPORTE, G., CHEN, Y., AND HE, R. An adaptive large neighborhood search metaheuristic for agile satellite scheduling with time-dependent transition time. *Computers & Operations Research 86* (2017), 41–53.

[102] LOUKOPOULOS, T., AND AHMAD, I. Static and adaptive distributed data replication using genetic algorithms. *Journal of Parallel and Distributed Computing 64*, 11 (2004), 1270–1285.

[103] MAN, K.-F., TANG, K.-S., AND KWONG, S. Genetic algorithms: concepts and applications [in engineering design]. *IEEE transactions on Industrial Electronics 43*, 5 (1996), 519–534.

[104] MANSOURI, Y., TOOSI, A. N., AND BUYYA, R. Cost optimization for dynamic replication and migration of data in cloud data centers. *IEEE Transactions on Cloud Computing 7*, 3 (2017), 705–718.

[105] MAO, H., ALIZADEH, M., MENACHE, I., AND KANDULA, S. Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks* (2016), ACM, pp. 50–56.

[106] MAO, M., AND HUMPHREY, M. A performance study on the vm startup time in the cloud. In *2012 IEEE Fifth International Conference on Cloud Computing* (2012), IEEE, pp. 423–430.

[107] MAO, Z., YANG, J., SHANG, Y., LIU, C., AND CHEN, J. A game theory of cloud service deployment. In *2013 IEEE Ninth World Congress on Services* (2013), IEEE, pp. 436–443.

[108] MARCHAND, H., MARTIN, A., WEISMANTEL, R., AND WOLSEY, L. Cutting planes in integer and mixed integer programming. *Discrete Applied Mathematics 123*, 1-3 (2002), 397–446.

[109] MENZEL, M., RANJAN, R., WANG, L., KHAN, S. U., AND CHEN, J. Cloudgenius: a hybrid decision support method for automating the migration of web application clusters to public clouds. *IEEE Transactions on Computers 64*, 5 (2014), 1336–1348.

[110] MEZURA-MONTES, E., AND COELLO, C. A. C. Constraint-handling in nature-inspired numerical optimization: past, present and future. *Swarm and Evolutionary Computation 1*, 4 (2011), 173–194.

[111] MICHALEWICZ, Z., DASGUPTA, D., LE RICHE, R. G., AND SCHOENAUER, M. Evolutionary algorithms for constrained engineering problems. *Computers & Industrial Engineering 30*, 4 (1996), 851–870.

[112] MICROSOFT. What is cloud computing? https://azure.microsoft.com/en-us/overview/what-is-cloud-computing/.

[113] MICROSOFT. Traffic manager routing methods. https://docs.microsoft.com/en-us/azure/traffic-manager/traffic-manager-routing-methods, 2021.

[114] MILANI, B. A., AND NAVIMIPOUR, N. J. A comprehensive review of the data replication techniques in the cloud environments: Major

trends and future directions. *Journal of Network and Computer Applications 64* (2016), 229–238.

[115] MIRJALILI, S. Genetic algorithm. In *Evolutionary algorithms and neural networks*. Springer, 2019, pp. 43–55.

[116] MITCHELL, J. E. Branch-and-cut algorithms for combinatorial optimization problems. *Handbook of applied optimization 1* (2002), 65–77.

[117] MNIH, V., BADIA, A. P., MIRZA, M., GRAVES, A., LILLICRAP, T., HARLEY, T., SILVER, D., AND KAVUKCUOGLU, K. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning* (2016), pp. 1928–1937.

[118] MNIH, V., KAVUKCUOGLU, K., SILVER, D., RUSU, A. A., VENESS, J., BELLEMARE, M. G., GRAVES, A., RIEDMILLER, M., FIDJELAND, A. K., OSTROVSKI, G., ET AL. Human-level control through deep reinforcement learning. *Nature 518*, 7540 (2015), 529.

[119] MOHAMED, M. F. Service replication taxonomy in distributed environments. *Service Oriented Computing and Applications 10*, 3 (2016), 317–336.

[120] MOLDOVAN, T. M., AND ABBEEL, P. Safe exploration in markov decision processes. *arXiv preprint arXiv:1205.4810* (2012).

[121] MYSQL. Eventual consistency. https://dev.mysql.com/doc/mysql-cluster-manager/1.4/en/mcm-eventual-consistency.html, 2021.

[122] NANDA, S., AND HACKER, T. J. Racc: Resource-aware container consolidation using a deep learning approach. In *Proceedings of the First Workshop on Machine Learning for Computing Systems* (2018), ACM, p. 2.

[123] NARDELLI, M., CARDELLINI, V., AND CASALICCHIO, E. Multi-level elastic deployment of containerized applications in geo-distributed environments. In *2018 IEEE 6th International Conference on Future Internet of Things and Cloud (FiCloud)* (2018), IEEE, pp. 1–8.

[124] NEWELL, C. *Applications of queueing theory*, vol. 4. Springer Science & Business Media, 2013.

[125] NIST. Cloud computing definitions. https://csrc.nist.gov/projects/cloud-computing.

[126] NOURI, S. M. R., LI, H., VENUGOPAL, S., GUO, W., HE, M., AND TIAN, W. Autonomic decentralized elasticity based on a reinforcement learning controller for cloud applications. *Future Generation Computer Systems 94* (2019), 765–780.

[127] OOI, B.-Y., CHAN, H.-Y., AND CHEAH, Y.-N. Dynamic service placement and replication framework to enhance service availability using team formation algorithm. *Journal of systems and Software 85*, 9 (2012), 2048–2062.

[128] PALADI, N., ASLAM, M., AND GEHRMANN, C. Trusted geolocation-aware data placement in infrastructure clouds. In *2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications* (2014), IEEE, pp. 352–360.

[129] PAREDIS, J. Co-evolutionary constraint satisfaction. In *International Conference on Parallel Problem Solving from Nature* (1994), Springer, pp. 46–55.

[130] PASZKE, A., GROSS, S., MASSA, F., LERER, A., BRADBURY, J., CHANAN, G., KILLEEN, T., LIN, Z., GIMELSHEIN, N., ANTIGA, L., ET AL. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems 32* (2019), 8026–8037.

[131] PETCU, D. Consuming resources and services from multiple clouds. *Journal of Grid Computing 12*, 2 (2014), 321–345.

[132] PISINGER, D., AND ROPKE, S. Large neighborhood search. In *Handbook of metaheuristics*. Springer, 2010, pp. 399–419.

[133] POWELL, D., AND SKOLNICK, M. M. Using genetic algorithms in engineering design optimization with non-linear constraints. In *Proceedings of the 5th International conference on Genetic Algorithms* (1993), pp. 424–431.

[134] QIN, Y., SONG, D., CHEN, H., CHENG, W., JIANG, G., AND COTTRELL, G. A dual-stage attention-based recurrent neural network for time series prediction. *arXiv preprint arXiv:1704.02971* (2017).

[135] QU, C., CALHEIROS, R. N., AND BUYYA, R. Auto-scaling web applications in clouds: A taxonomy and survey. *ACM Computing Surveys (CSUR) 51*, 4 (2018), 1–33.

[136] RAICU, I., FOSTER, I. T., AND ZHAO, Y. Many-task computing for grids and supercomputers. In *2008 workshop on many-task computing on grids and supercomputers* (2008), IEEE, pp. 1–11.

[137] RAMPÉREZ, V., SORIANO, J., LIZCANO, D., ALJAWARNEH, S., AND LARA, J. A. From SLA to vendor-neutral metrics: An intelligent knowledge-based approach for multi-cloud sla-based broker. *International Journal of Intelligent Systems* (2021), DOI: 10.1002/int.22638.

[138] RANCHER. Hybrid cloud and multi cloud. https://rancher.com, 2021.

[139] RODOLAKIS, G., SIACHALOU, S., AND GEORGIADIS, L. Replicated server placement with qos constraints. *IEEE Transactions on Parallel and Distributed Systems 17*, 10 (2006), 1151–1162.

[140] ROSSI, F., CARDELLINI, V., PRESTI, F. L., AND NARDELLI, M. Geo-distributed efficient deployment of containers with kubernetes. *Computer Communications 159* (2020), 161–174.

[141] ROSSI, F., NARDELLI, M., AND CARDELLINI, V. Horizontal and vertical scaling of container-based applications using reinforcement learning. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)* (2019), IEEE, pp. 329–338.

[142] SABHARWAL, N., AND WALI, P. Cloud capacity management. In *Cloud Capacity Management*. Springer, 2013, pp. 35–54.

[143] SAHNI, J., AND VIDYARTHI, D. P. A cost-effective deadline-constrained dynamic scheduling algorithm for scientific workflows in a cloud environment. *IEEE Transactions on Cloud Computing 6*, 1 (2018), 2–18.

[144] SAIDU, I., SUBRAMANIAM, S., JAAFAR, A., AND ZUKARNAIN, Z. A. A load-aware weighted round-robin algorithm for ieee 802.16 networks. *EURASIP Journal on Wireless Communications and Networking 2014*, 1 (2014), 1–12.

[145] SAKELLARIOU, R., ZHAO, H., TSIAKKOURI, E., AND DIKAIAKOS, M. D. Scheduling workflows with budget constraints. In *Integrated research in GRID computing*. Springer, 2007, pp. 189–202.

[146] SALCEDO-SANZ, S. A survey of repair methods used as constraint handling techniques in evolutionary algorithms. *Computer science review 3*, 3 (2009), 175–192.

[147] SCHOENAUER, M., AND XANTHAKIS, S. Constrained ga optimization. In *Proc. 5th International Conference on Genetic Algorithms* (1993), Morgan Kaufmann, pp. 573–580.

[148] SCHULMAN, J., LEVINE, S., ABBEEL, P., JORDAN, M., AND MORITZ, P. Trust region policy optimization. In *International Conference on Machine Learning* (2015), pp. 1889–1897.

[149] SHAW, P. Using constraint programming and local search methods to solve vehicle routing problems. In *International conference on principles and practice of constraint programming* (1998), Springer, pp. 417–431.

[150] SHI, T., CHEN, G., AND MA, H. Multi-objective container consolidation in cloud data centers. In *Australasian Joint Conference on Artificial Intelligence* (2018), Springer, pp. 783–795.

[151] SHI, T., MA, H., AND CHEN, G. Energy-aware container consolidation based on pso in cloud data centers. In *2018 IEEE Congress on Evolutionary Computation (CEC)* (2018), IEEE, pp. 1–8.

[152] SHI, T., MA, H., AND CHEN, G. A genetic-based approach to location-aware cloud service brokering in multi-cloud environment. In *2019 IEEE International Conference on Services Computing (SCC)* (2019), IEEE, pp. 146–153.

[153] SHI, T., MA, H., AND CHEN, G. A seeding-based GA for location-aware workflow deployment in multi-cloud environment. In *2019 IEEE Congress on Evolutionary Computation (CEC)* (2019), IEEE, pp. 3364–3371.

[154] SHI, T., MA, H., AND CHEN, G. Divide and conquer: Seeding strategies for multi-objective multi-cloud composite applications deployment. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion* (2020), pp. 317–318.

[155] SHI, T., MA, H., AND CHEN, G. Seeding-based multi-objective evolutionary algorithms for multi-cloud composite applications de-

ployment. In *2020 IEEE International Conference on Services Computing (SCC)* (2020), IEEE, pp. 240–247.

[156] SHI, T., MA, H., CHEN, G., AND HARTMANN, S. Location-aware and budget-constrained service deployment for composite applications in multi-cloud environment. *IEEE Transactions on Parallel and Distributed Systems 31*, 8 (2020), 1954–1969.

[157] SHI, T., MA, H., CHEN, G., AND HARTMANN, S. Cost-effective web application replication and deployment in multi-cloud environment. *IEEE Transactions on Parallel and Distributed Systems 33*, 8 (2021), 1982–1995.

[158] SHI, T., MA, H., CHEN, G., AND HARTMANN, S. Location-aware and budget-constrained service brokering in multi-cloud via deep reinforcement learning. In *International Conference on Service-Oriented Computing* (2021), Springer, pp. 756–764.

[159] SHI, T., MA, H., CHEN, G., AND SVEN, H. Location-aware and budget-constrained application replication and deployment in multi-cloud environment. In *IEEE International Conference on Web Services (ICWS 2020)* (2020), IEEE, pp. 110–117.

[160] SHI, W., CAO, J., ZHANG, Q., LI, Y., AND XU, L. Edge computing: Vision and challenges. *IEEE internet of things journal 3*, 5 (2016), 637–646.

[161] SIAMI-NAMINI, S., TAVAKOLI, N., AND NAMIN, A. S. A comparison of arima and lstm in forecasting time series. In *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)* (2018), IEEE, pp. 1394–1401.

[162] SILVER, D., HUANG, A., MADDISON, C. J., GUEZ, A., SIFRE, L., VAN DEN DRIESSCHE, G., SCHRITTWIESER, J., ANTONOGLOU, I., PANNEERSHELVAM, V., LANCTOT, M., ET AL. Mastering the game

of go with deep neural networks and tree search. *nature 529*, 7587 (2016), 484.

[163] SKYTAP. Accessing VMs with published services. https://help.skytap.com/accessing-vms-with-published-services.html, 2019.

[164] SLIMANI, S., HAMROUNI, T., AND BEN CHARRADA, F. Service-oriented replication strategies for improving quality-of-service in cloud computing: a survey. *Cluster Computing* (2020), 1–32.

[165] SMANCHAT, S., AND VIRIYAPANT, K. Taxonomies of workflow scheduling problem and techniques in the cloud. *Future Generation Computer Systems 52* (2015), 1–12.

[166] SMITH, A. E., COIT, D. W., BAECK, T., FOGEL, D., AND MICHALEWICZ, Z. Penalty functions. *Handbook of evolutionary computation 97*, 1 (1997), C5.

[167] SRIRAMA, S. N., ADHIKARI, M., AND PAUL, S. Application deployment using containers with auto-scaling for microservices in cloud environment. *Journal of Network and Computer Applications 160* (2020), 102629.

[168] STAMFORD, C. Gartner forecasts worldwide public cloud revenue to grow 17.3 percent in 2019. https://www.gartner.com/en/newsroom/press-releases/2018-09-12-gartner-forecasts-worldwide-public-cloud-revenue-to-grow-17-percent-in-2019, 2018.

[169] SURRY, P. D., RADCLIFFE, N. J., ET AL. The comoga method: constrained optimisation by multi-objective genetic algorithms. *Control and Cybernetics 26* (1997), 391–412.

[170] SUTTON, R. S., AND BARTO, A. G. *Reinforcement learning: An introduction*. MIT press, 2018.

[171] TAN, B., MA, H., MEI, Y., AND ZHANG, M. Evolutionary multi-objective optimization for web service location allocation problem. *IEEE Transactions on Services Computing 14*, 2 (2018), 458–471.

[172] TANG, X. Reliability-aware cost-efficient scientific workflows scheduling strategy on multi-cloud systems. *IEEE Transactions on Cloud Computing* (2021), DOI: 10.1109/TCC.2021.3057422.

[173] TANG, Z., ZHOU, X., ZHANG, F., JIA, W., AND ZHAO, W. Migration modeling and learning algorithms for containers in fog computing. *IEEE Transactions on Services Computing 12*, 5 (2018), 712–725.

[174] TECHNOLOGIES, D. Iaas vs paas solutions. https://www.delltechnologies.com/en-nz/learn/cloud/iaas-vs-paas.htm, 2021.

[175] TESAURO, G., JONG, N. K., DAS, R., AND BENNANI, M. N. A hybrid reinforcement learning approach to autonomic resource allocation. In *2006 IEEE International Conference on Autonomic Computing* (2006), IEEE, pp. 65–73.

[176] TESSEMA, B., AND YEN, G. G. A self adaptive penalty function based algorithm for constrained optimization. In *2006 IEEE international conference on evolutionary computation* (2006), IEEE, pp. 246–253.

[177] TOKA, L., DOBREFF, G., FODOR, B., AND SONKOLY, B. Adaptive ai-based auto-scaling for kubernetes. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)* (2020), IEEE, pp. 599–608.

[178] TOOSI, A. N., CALHEIROS, R. N., AND BUYYA, R. Interconnected cloud computing environments: Challenges, taxonomy, and survey. *ACM Computing Surveys (CSUR) 47*, 1 (2014), 7.

[179] TOOSI, A. N., QU, C., DE ASSUNÇÃO, M. D., AND BUYYA, R. Renewable-aware geographical load balancing of web applications for sustainable data centers. *Journal of Network and Computer Applications 83* (2017), 155–168.

[180] TOPCUOGLU, H., HARIRI, S., AND WU, M.-Y. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE transactions on parallel and distributed systems 13*, 3 (2002), 260–274.

[181] TORDSSON, J., MONTERO, R. S., MORENO-VOZMEDIANO, R., AND LLORENTE, I. M. Cloud brokering mechanisms for optimized placement of virtual machines across multiple providers. *Future Generation Computer Systems 28*, 2 (2012), 358–367.

[182] TRAN, K.-T., AND AGOULMINE, N. Adaptive and cost-effective service placement. In *2011 IEEE Global Telecommunications Conference-GLOBECOM 2011* (2011), IEEE, pp. 1–6.

[183] TSITSIKLIS, J. N., AND VAN ROY, B. Analysis of temporal-diffference learning with function approximation. In *Advances in neural information processing systems* (1997), pp. 1075–1081.

[184] VAN BAAREN, E.-J. Wikibench: A distributed, wikipedia based web application benchmark. *Master's thesis, VU University Amsterdam* (2009).

[185] VAN LAARHOVEN, P. J., AND AARTS, E. H. Simulated annealing. In *Simulated annealing: Theory and applications*. Springer, 1987, pp. 7–15.

[186] VARANGAONKAR, A. Types of cloud computing services: Iaas, paas, and saas. https://hub.packtpub.com/cloud-computing-services-iaas-paas-saas/, 2018.

[187] VILAPLANA, J., SOLSONA, F., TEIXIDÓ, I., MATEO, J., ABELLA, F., AND RIUS, J. A queuing theory model for cloud computing. *The Journal of Supercomputing 69*, 1 (2014), 492–507.

[188] VMWARE. Maximizing the value of multi-cloud. https://www.vmware.com/content/microsites/possible/stories/maximizing-the-value-of-multi-cloud.html, 2022.

[189] VOGELS, W. Eventually consistent. *Communications of the ACM 52*, 1 (2009), 40–44.

[190] VOUDOURIS, C., AND TSANG, E. Guided local search and its application to the traveling salesman problem. *European journal of operational research 113*, 2 (1999), 469–499.

[191] VOUDOURIS, C., TSANG, E. P., AND ALSHEDDY, A. Guided local search. In *Handbook of metaheuristics*. Springer, 2010, pp. 321–361.

[192] WANG, L. Architecture-based reliability-sensitive criticality measure for fault-tolerance cloud applications. *IEEE Transactions on Parallel and Distributed Systems 30*, 11 (2019), 2408–2421.

[193] WANG, S., DING, Z., AND JIANG, C. Elastic scheduling for microservice applications in clouds. *IEEE Transactions on Parallel and Distributed Systems 32*, 1 (2020), 98–115.

[194] WATKINS, C. J., AND DAYAN, P. Q-learning. *Machine learning 8*, 3-4 (1992), 279–292.

[195] WEN, Z., CAŁA, J., WATSON, P., AND ROMANOVSKY, A. Cost effective, reliable and secure workflow deployment over federated clouds. *IEEE Transactions on Services Computing 10*, 6 (2017), 929–941.

[196] WEYNS, D., SCHMERL, B., GRASSI, V., MALEK, S., MIRANDOLA, R., PREHOFER, C., WUTTKE, J., ANDERSSON, J., GIESE, H., AND

GÖSCHKA, K. M. On patterns for decentralized control in self-adaptive systems. In *Software Engineering for Self-Adaptive Systems II*. Springer, 2013, pp. 76–107.

[197] WHITLEY, D. A genetic algorithm tutorial. *Statistics and computing 4*, 2 (1994), 65–85.

[198] WICKREMASINGHE, B., CALHEIROS, R. N., AND BUYYA, R. Cloudanalyst: A cloudsim-based visual modeller for analysing cloud computing environments and applications. In *Advanced Information Networking and Applications (AINA), 2010 24th IEEE International Conference on* (2010), IEEE, pp. 446–452.

[199] WU, F., WU, Q., AND TAN, Y. Workflow scheduling in cloud: a survey. *The Journal of Supercomputing 71*, 9 (2015), 3373–3418.

[200] WU, J., ZHANG, B., YANG, L., WANG, P., AND ZHANG, C. A replicas placement approach of component services for service-based cloud application. *Cluster Computing 19*, 2 (2016), 709–721.

[201] WU, Y., WU, C., LI, B., ZHANG, L., LI, Z., AND LAU, F. C. Scaling social media applications into geo-distributed clouds. *IEEE/ACM Transactions On Networking 23*, 3 (2014), 689–702.

[202] XIA, X., CHEN, F., GRUNDY, J., ABDELRAZEK, M., JIN, H., AND HE, Q. Constrained app data caching over edge server graphs in edge computing environment. *IEEE Transactions on Services Computing* (2021), DOI: 10.1109/TSC.2021.3062017.

[203] XIA, Y., ZHOU, M., LUO, X., ZHU, Q., LI, J., AND HUANG, Y. Stochastic modeling and quality evaluation of infrastructure-as-a-service clouds. *IEEE Transactions on Automation Science and Engineering 12*, 1 (2013), 162–170.

[204] XIAO, L., JOHANSSON, M., AND BOYD, S. P. Simultaneous routing and resource allocation via dual decomposition. *IEEE Transactions on Communications 52*, 7 (2004), 1136–1144.

[205] XU, R., WANG, Y., LUO, H., WANG, F., XIE, Y., LIU, X., AND YANG, Y. A sufficient and necessary temporal violation handling point selection strategy in cloud workflow. *Future Generation Computer Systems 86* (2018), 464–479.

[206] YENIAY, Ö. Penalty function methods for constrained optimization with genetic algorithms. *Mathematical and computational Applications 10*, 1 (2005), 45–56.

[207] YI, D., ZHOU, X., WEN, Y., AND TAN, R. Efficient compute-intensive job allocation in data centers via deep reinforcement learning. *IEEE Transactions on Parallel and Distributed Systems 31*, 6 (2020), 1474–1485.

[208] YU, J., AND BUYYA, R. A budget constrained scheduling of workflow applications on utility grids using genetic algorithms. In *2006 Workshop on Workflows in Support of Large-Scale Science* (2006), IEEE, pp. 1–10.

[209] ZENG, L., VEERAVALLI, B., AND LI, X. Scalestar: Budget conscious scheduling precedence-constrained many-task workflow applications in cloud. In *2012 IEEE 26th International Conference on Advanced Information Networking and Applications* (2012), IEEE, pp. 534–541.

[210] ZHANG, C., XU, Y., HU, Y., WU, J., REN, J., AND ZHANG, Y. A blockchain-based multi-cloud storage data auditing scheme to locate faults. *IEEE Transactions on Cloud Computing* (2021), DOI: 10.1109/TCC.2021.3057771.

[211] ZHANG, M., LIU, L., AND LIU, S. Genetic algorithm based QoS-aware service composition in multi-cloud. In *2015 IEEE Conference on Collaboration and Internet Computing (CIC)* (2015), IEEE, pp. 113–118.

[212] ZHANG, Q., ZHU, Q., ZHANI, M. F., BOUTABA, R., AND HELLERSTEIN, J. L. Dynamic service placement in geographically distributed clouds. *IEEE Journal on Selected Areas in Communications 31*, 12 (2013), 762–772.

[213] ZHANG, S., WU, T., PAN, M., ZHANG, C., AND YU, Y. A-sarsa: A predictive container auto-scaling algorithm based on reinforcement learning. In *2020 IEEE International Conference on Web Services (ICWS)* (2020), IEEE, pp. 489–497.

[214] ZHANG, Y., DI, B., ZHENG, Z., LIN, J., AND SONG, L. Distributed multi-cloud multi-access edge computing by multi-agent reinforcement learning. *IEEE Transactions on Wireless Communications 20*, 4 (2020), 2565–2578.

[215] ZHANG, Y., YAO, J., AND GUAN, H. Intelligent cloud resource management with deep reinforcement learning. *IEEE Cloud Computing 4*, 6 (2017), 60–69.

[216] ZHENG, W., AND SAKELLARIOU, R. Budget-deadline constrained workflow planning for admission control. *Journal of grid computing 11*, 4 (2013), 633–651.

[217] ZHONG, Z., AND BUYYA, R. A cost-efficient container orchestration strategy in kubernetes-based cloud computing infrastructures with heterogeneous resources. *ACM Transactions on Internet Technology (TOIT) 20*, 2 (2020), 1–24.

[218] ZHU, X., VANDEN BROUCKE, S., ZHU, G., VANTHIENEN, J., AND BAESENS, B. Enabling flexible location-aware business process modeling and execution. *Decision Support Systems 83* (2016), 1–9.

[219] ZHU, Z., ZHANG, G., LI, M., AND LIU, X. Evolutionary multi-objective workflow scheduling in cloud. *IEEE Transactions on parallel and distributed Systems 27*, 5 (2015), 1344–1357.