Auto-scaling and Deployment of Web Applications in Distributed Computing Clouds

Chenhao Qu

Submitted in total fulfilment of the requirements of the degree of

Doctor of Philosophy

Department of Computing and Information Systems
THE UNIVERSITY OF MELBOURNE

Dec. 2016

Auto-scaling and Deployment of Web Applications in Distributed Computing Clouds

Chenhao Qu
Principal Supervisor: Prof. Rajkumar Buyya
Co-Supervisor: Dr. Rodrigo N. Calheiros

Abstract

CLOUD Computing, which allows users to acquire/release resources based on real-time demand from large data centers in a pay-as-you-go model, has attracted considerable attention from the ICT industry. Many web application providers have moved or plan to move their applications to Cloud, as it enables them to focus on their core business by freeing them from the task and the cost of managing their data center infrastructures, which are often over-provisioned or under-provisioned under a dynamic workload.

Applications these days commonly serve customers from geographically dispersed regions. Therefore, to meet the stringent Quality of Service (QoS) requirements, they have to be deployed in multiple data centers close to the end customer locations. However, efficiently utilizing Cloud resources to reach high cost-efficiency, low network latency, and high availability is a challenging task for web application providers, especially when the service provider intends to deploy the application in multiple geographical distributed Cloud data centers. The problems, including how to identify satisfactory Cloud offerings, how to choose geographical locations of data centers so that the network latency is minimized, how to provision the application with minimum cost incurred, and how to guarantee high availability under failures and flash crowds, should be addressed to enable QoS-aware and cost-efficient utilization of Cloud resources.

In this thesis, we investigated techniques and solutions for these questions to help application providers to efficiently manage deployment and provision of their applications in distributed computing Clouds. It extended the state-of-the-art by making the following contributions:

1. A hierarchical fuzzy inference approach for identifying satisfactory Cloud services according to individual requirements.

2. Algorithms for selection of multi-Cloud data centers and deployment of applications on them to minimize Service Level Objective (SLO) violations for web applications requiring strong consistency.

3. An auto-scaler for web applications that achieves both high availability and significant cost saving by using heterogeneous spot instances.

4. An approach that mitigates the impact of short-term application overload caused by either resource failures or flash crowds in any individual data center through geographical load balancing.

Declaration

This is to certify that

1. the thesis comprises only my original work towards the PhD,
2. due acknowledgement has been made in the text to all other material used,
3. the thesis is less than 100,000 words in length, exclusive of tables, maps, bibliographies and appendices.

Chenhao Qu, Dec. 2016

Acknowledgements

Throughout my rewarding but challenging Ph.D. journey, my supervisor Professor Rajkumar Buyya has provided me insightful guidance and continuous support, without which this could never happen. Besides, my co-supervisor Dr. Rodrigo N. Calheiros has generously offered me his knowledge and technical assistance, which are indispensable to the completion of this thesis. I sincerely thank them for their kindness and excellent contributions. I also want to express my gratitude to Professor Zhen Xiao who gave me his valuable advice as my external supervisor.

I would like to thank the chair of my Ph.D. committee Professor Lars Kulik for his support. I also thank Dr. Adel Nadjaran Toosi and Dr. Amir Vahid Dastjerdi for the discussions.

I am grateful to all the current and past members of the CLOUDS Laboratory, at the University of Melbourne: Maria Rodriguez, Atefeh Khosravi, Sareh Fotuhi, Yaser Mansouri, Deepak Poola, Yali Zhao, Jungmin Jay Son, Bowen Zhou, Farzad Khodadadi, Deborah Magalhes, Tiago Justino, Safiollah Heidari, Liu Xunyun, Nitisha Jain, Saurabh Garg, William Voorsluys, Patricia Arroba Garca, Caesar Wu, Minxian Xu, Sara Kardani Moghaddam, Muhammad H. Hilman, and Redowan Mahmud, for their friendship.

I thank the University of Melbourne for providing the financial support and appropriate facilities to let me pursue my Ph.D. study. I also appreciate the support from the Department of Computing and Information Systems. I would like to thank the administrative staff members Rhonda Smithies, Madalain Dolic, and Julie Ireland, and the Head of the Department Professor Justin Zobel for their work.

Finally, I would like to thank my parents who have encouraged me to overcome all the difficulties encountered in this journey successfully. Their love and care are indispensable to me and my achievements.

Chenhao Qu

Melbourne, Australia

Preface

This thesis research has been carried out in the Cloud Computing and Distributed Systems (CLOUDS) Laboratory, Department of Computing and Information Systems, The University of Melbourne. The main contributions of the thesis are discussed in Chapters 2 — 6 and are based on the following publications:

- **Chenhao Qu**, Rodrigo N. Calheiros, and Rajkumar Buyya, "A Taxonomy and Survey of Auto-scaling Web Applications in Clouds", *ACM Computing Surveys* (Under Review), 2016.
- **Chenhao Qu**, and Rajkumar Buyya, "A Cloud Trust Evaluation System using Hierarchical Fuzzy Inference System for Service Selection", *Proceedings of the 28th IEEE International Conference on Advanced Information Networking and Applications (AINA 2014, IEEE CS Press, USA)*, Victoria, Canada, May 13-16, 2014.
- **Chenhao Qu**, Rodrigo N. Calheiros, and Rajkumar Buyya, "SLO-aware Deployment of Web Applications Requiring Strong Consistency using Multiple Clouds", *Proceedings of the 8th IEEE International Conference on Cloud Computing (Cloud 2015, IEEE CS Press, USA)*, New York, USA, June 27 - July 2, 2015.
- **Chenhao Qu**, Rodrigo N. Calheiros, and Rajkumar Buyya, "A Reliable and Cost-Efficient Auto-Scaling System for Web Applications Using Heterogeneous Spot Instances", *Journal of Network and Computer Applications*, issue 65, page 167 - 180, 2016.
- **Chenhao Qu**, Rodrigo N. Calheiros, and Rajkumar Buyya, "Mitigating Impact of Short-term Overload on Multi-Cloud Web Applications through Geographical Load Balancing", accepted by *Concurrency and Computation Practice and Experience*, 2017.

Contents

List of Figures

List of Tables

Chapter 1
Introduction

CLOUD Computing offers services from large data centers to end users. It adopts a so-called pay-as-you-go model in which customers acquire services or resources based on their demand and are only charged for their actual usage, which realizes the long-held dream of making computing the fifth utility [28]. Due to economies of scale and multiplexing, since its birth, Cloud computing has been creating substantial monetary values for both Cloud providers and Cloud customers. From the users' perspective, it not only gives them an option to avoid investing in costly proprietary data center infrastructures and their maintenance to reduce the IT expenditure, but also allows them to fully focus on their main business in this rapid-growing Internet era [120], in which more than ever fast delivering and growth are essential to survive and prosperity.

Base on the services provided, NIST classified Cloud computing into three categories [141] as shown in Figure 1.1 : 1) Infrastructure as a Service (IaaS) which leases computing resources, such as processing, storage, and network, 2) Platform as a Service (PaaS) that offers services to host and manage applications, and 3) Software as a Service (SaaS) which directly provides applications to the end customers. This thesis focuses on IaaS, and in the following content, the term Cloud refers to IaaS only.

As a general purpose computing platform, Clouds can host a broad spectrum of applications, such as scientific simulations, data analytics, and interactive web applications. Web applications are traditionally hosted in either proprietary infrastructures or rented server rooms. In addition to the high upfront investment cost, these approaches are difficult to scale, leaving applications either over-provisioned or under-provisioned under a dynamic workload. The elasticity feature of Clouds that allows users to acquire/release

Figure 1.1: Cloud computing service model

a virtually unlimited amount of resources dynamically makes Cloud the ideal platform to host web applications whose workload are fluctuant in nature. Due to its advantages compared to traditional solutions, many organizations providing web applications, from e-commerce business *ebay* [57] to US government [13], have shifted or been moving their applications to Cloud.

As Cloud providers are building more data centers around the world increasingly, it opens the opportunity for web application providers to utilize this globally available Cloud infrastructure to boost the performance of their applications. In summary, deploying web applications in multiple geographically distributed data centers brings the following significant benefits: 1) it improves availability of applications even under unexpected data center outages, 2) it provides balanced and satisfactory Quality of Service (QoS) to geographically dispersed customers, 3) it helps to comply with data regulations, 4) it avoids vendor lock-in, and 5) it enables cost optimization among different vendors. However, efficiently and effectively utilizing multiple Clouds to achieve the above benefits remains a challenging task.

Figure 1.2 illustrates a typical scenario of a web application running on geographically distributed Clouds. Among all the available Cloud data centers, service providers

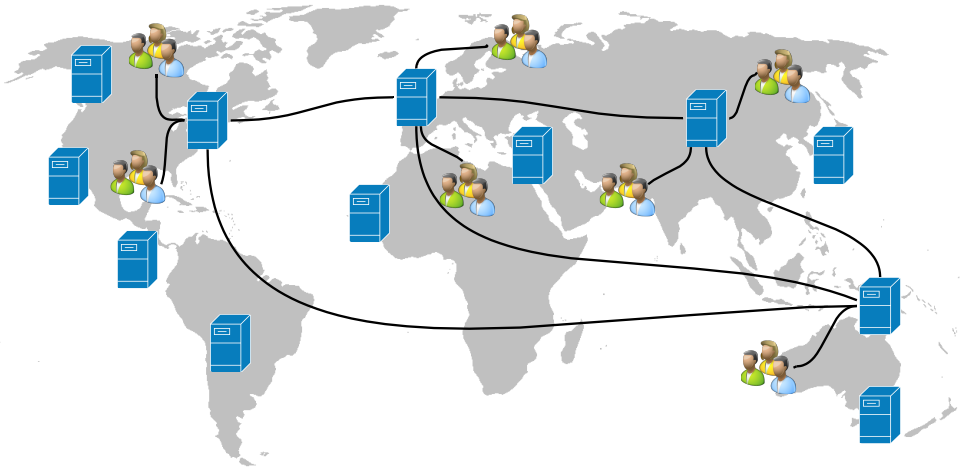


Figure 1.2: A web application deployed on geographically distributed Clouds

deploy their applications on a subset of data centers chosen in consideration of user locations, interactions between the selected data centers, and other factors. The users of the application always send requests to the closest data centers to obtain better QoS. To manage applications in this scenario, application provider needs to tackle many challenges which we will discuss below.

1.1 Challenges in Managing Applications in Distributed Computing Clouds

To manage application in an environment composed of geographically distributed Cloud data centers, we consider two major aspects: 1) deployment, and 2) provision. We define deployment as the process to choose the right data centers to host the application. While provision involves acquiring resources from the selected providers during runtime to ensure the application has enough resources to fulfill the task and achieve satisfactory QoS. The significant challenges involved in these aspects are noted in Figure 1.3.

1.1.1 Challenges in Deployment

The deployment aspect faces the challenges of:

- **Identifying Suitable Services:** the application provider needs to discover services

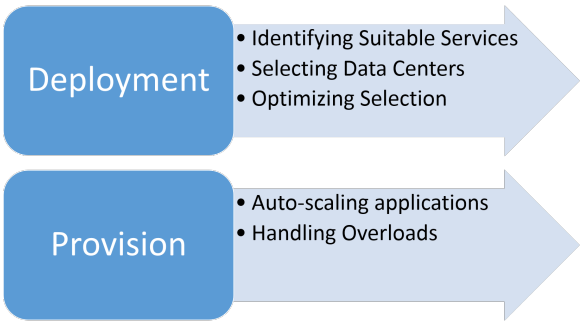


Figure 1.3: The challenges of managing web applications in multiple-Clouds

that can satisfy the functional and QoS requirements of the application among many listed services.

- **Selecting Data Centers:** after suitable services have been identified, the application provider selects a set of them to host the application in consideration of achievable QoS according to their end user locations, cost-efficiency, regulation compliance, and other factors.

- **Optimizing Selection:** during operation of the application, the application provider may face changing workload pattern and business expansion. In these cases, the selection of data centers should be optimized to cater for these changes. Such optimizations also need to take migration cost of the application into account when making the decisions.

1.1.2 Challenges in Provision

The provision aspect involves the following challenges:

- **Auto-scaling Applications:** during running time, application provider should dynamically acquire/release resources to match the fluctuant incoming workload to meet the QoS requirements with minimum cost.

- **Handling Overloads:** inevitably, sometimes the incoming workload may exceed the capacity of available resources either because of resource failures or sudden

surges of the workload. The application provider needs to prepare for these situations to minimize degradation of the application performance.

1.2 Research Problems and Objectives

The target of this thesis is to explore the deployment and provision aspects of web application management across multiple Clouds. It can be summarized by the following research question:

How to efficiently select Cloud services for hosting web applications across multiple geographically distributed Cloud data centers and provision resources under dynamic workload considering requirements including:

- Infrastructure Requirement — the Cloud provider should satisfy the needs of the application regarding its functional and infrastructure level Quality of Service (QoS) requirements.
- Network Latency Requirement — deployment of the application should be dispersed to data centers that will minimize the perceived network latencies by end users.
- Cost Requirement — minimum resources should be provisioned to the application under a dynamic workload to ensure the latency threshold is met with as less financial cost as possible.
- Violation-over Requirement — SLA violations should be timely detected and effectively dealt with.

To explore these problems, we first need to have a technique that can identify satisfactory Cloud services. It can work with general-purpose applications rather than catering only for web applications. When implementing such method, it raises several questions:

- How to efficiently evaluate all the available services?
- How to make it easy to use without sacrificing accuracy?

- How to comprehensively take all the involved QoS attributes into account?
- How to consider instability of performances when evaluating the services?

Secondly, we have to select a set of data centers to host the application, which needs to address the following challenges:

- How to minimize network latencies perceived by end users that are geographically dispersed?
- How to model latencies if strong consistency is required across data centers?
- How to efficiently find an acceptable solution for the often NP-hard QoS-aware data center selection problem?
- How to reach a balance of resource cost and performance?
- How to optimize the selection considering migration cost under dynamic workload?

After deployment of the application, provision should be managed to ensure enough resources are available to the application while incurring a minimum cost. To realize that, the following issues need to be tackled:

- How to estimate resource consumption?
- How to predict incoming workload?
- How to self-adapt the provision in case of changing application and workload?
- How to reduce provision oscillation?
- How to determine resource combination and how to provision with minimum cost?
- How to timely detect and handle SLO violations?

1.3 Evaluation Methodology

The approaches in this thesis were evaluated using traces from real applications [204] or synthetic ones derived according to the study of their characteristics when no corresponding real trace is available. We used TPC-W [195], a transactional e-commerce application, Twissandra [196], a Twitter-like NoSQL-backed social network application, and Wikibench [203], an open source Wikipedia, as benchmark applications in different chapters of the thesis. TPC-W and Twissandra are employed to create realistic simulation scenarios for testing. Wikibench is deployed on Clouds to conduct real experiments of the prototype implementations.

Simulations are adopted to evaluate our approaches when repeatable large-scale experiments are not feasible or costly and difficult to conduct. In these cases, corresponding simulation tools are developed or extended and realistic settings are used for experiments.

Proof-of-concept experiments executed on real platforms are conducted to illustrate the feasibility and efficacy of the proposed approaches in Chapter 5 and Chapter 6. We have developed prototypes and automation scripts to enable repetition of our experiments, which we release for the convenience of the community [1]. Results of our proposed approaches are compared with industry solutions. Baselines have also been implemented and automated to facilitate duplication of experiments and results.

1.4 Thesis Contribution

The thesis has made the following contributions to answer the defined research question:

1. A taxonomy and survey of the state-of-the-art Cloud service selection and automatic provisioning techniques for web applications.

2. A technique that evaluates and ranks the satisfiability of Cloud services against user's individual QoS requirements using hierarchical fuzzy inference system.

    - Architecture definition of the framework regarding their interactions.

[1] https://github.com/quchenhao/

- A hierarchical fuzzy inference approach that
  - comprehensively evaluates a Cloud service in accordance to the hierarchical metrics of Cloud services;
  - supports using linguistic terms to define users' QoS requirements and preferences of metrics;
  - considers performance variations of the services using statistical metrics when assessing their QoS.

3. An approach that selects Cloud data centers to host web applications requiring strong consistency and dynamically optimizes the deployment to minimize violations of Service Level Objectives (SLO).

   - Definition of the architecture and its components.
   - A latency model for web applications.
   - Consistency latency models for Cassandra and Galera Cluster.
   - A genetic-based algorithm to select data centers for hosting web applications to minimize SLO violations.
   - A decision-making algorithm to optimize the selection of data centers under dynamic workload considering both SLO violations and migration cost.

4. An auto-scaler for web applications using heterogeneous spot instances to reach both high availability and low resource cost.

   - A fault-tolerant model for auto-scaling using heterogeneous spot instances to reach both high availability and superior cost-efficiency.
   - Algorithms and optimizations that comply semantics of the fault-tolerant model for scaling up and scaling down.
   - Prototype implementations both on a real Cloud and a simulation platform.

5. A technique that handles short-term overload events in any data center through geographical load balancing.

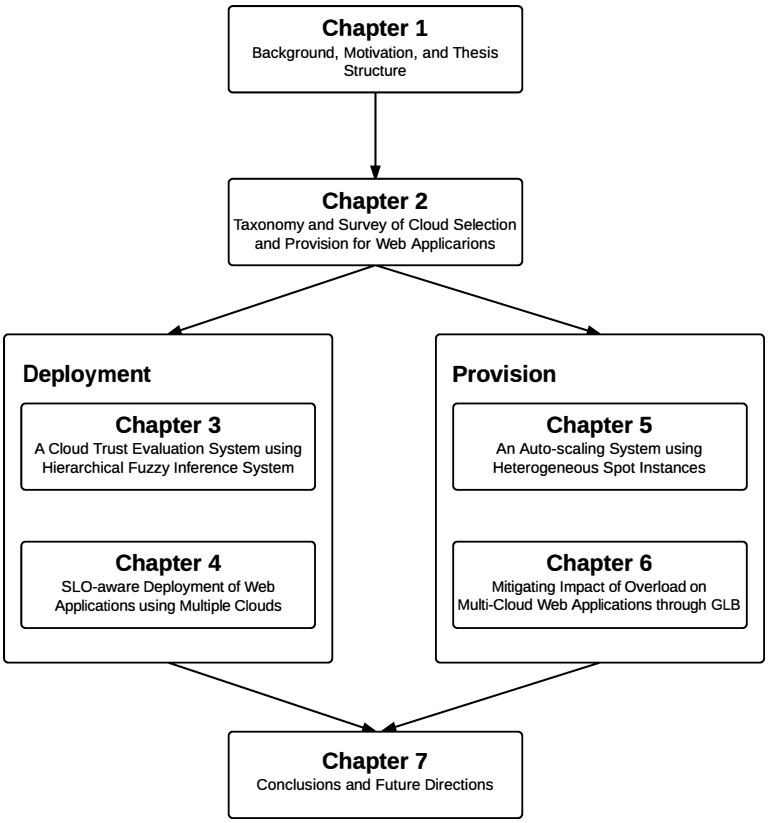


Figure 1.4: Thesis organization

- A decentralized geographical load balancing architecture and interactions among the components.
- An overload detection algorithm and an overload handling algorithm.
- A queuing model for deciding load redirection among data centers with available capacities to minimize overall latency increase.
- A prototype implementation of the proposed approach.

1.5 Thesis Organization

Figure 1.4 depicts the structure of the thesis and logic dependencies between the chapters. Chapter 2 presents a taxonomy and survey of the related literature. Chapters 3 and 4 focus on deployment of web applications. Chapters 5 and 6 target the challenges in provision aspect of web application management in Clouds. More specifically, the remainder

of the thesis are organized as follows:

- Chapter 2 presents a taxonomy and survey of selection Cloud services and provisioning resources for web applications in Clouds. It is partially derived from:
  - **Chenhao Qu**, Rodrigo N. Calheiros, and Rajkumar Buyya, "A Taxonomy and Survey of Auto-scaling Web Applications in Clouds", *ACM Computing Surveys* (Under Review), 2016.
- Chapter 3 proposes a technique that evaluates the satisfiability of Cloud services against user's individual QoS requirements. It is derived from:
  - **Chenhao Qu**, and Rajkumar Buyya, "A Cloud Trust Evaluation System using Hierarchical Fuzzy Inference System for Service Selection", *Proceedings of the 28th IEEE International Conference on Advanced Information Networking and Applications (AINA 2014, IEEE CS Press, USA)*, Victoria, Canada, May 13-16, 2014.
- Chapter 4 describes a mechanism that selects data centers according to end users' locations to minimize SLO violations for web applications requiring strong consistency and then optimizes the selection under a dynamic workload. It is derived from:
  - **Chenhao Qu**, Rodrigo N. Calheiros, and Rajkumar Buyya, "SLO-aware Deployment of Web Applications Requiring Strong Consistency using Multiple Clouds", *Proceedings of the 8th IEEE International Conference on Cloud Computing (Cloud 2015, IEEE CS Press, USA)*, New York, USA, June 27 - July 2, 2015.
- Chapter 5 proposes an auto-scaler for web applications using heterogeneous spot instances to achieve both high availability and superior cost-efficiency. It is derived from:
  - **Chenhao Qu**, Rodrigo N. Calheiros, and Rajkumar Buyya, " A Reliable and Cost-Efficient Auto-Scaling System for Web Applications Using Heterogeneous Spot Instances", *Journal of Network and Computer Applications*, issue 65, page 167 - 180, 2016.

- Chapter 6 introduces an approach that detects short-term overload situations in any participating data center and handles them through geographical load balancing. It is derived from:

  - **Chenhao Qu**, Rodrigo N. Calheiros, and Rajkumar Buyya, "Mitigating Impact of Short-term Overload on Multi-Cloud Web Applications through Geographical Load Balancing", accepted by *Concurrency and Computation Practice and Experience*, 2017.

- Chapter 7 concludes findings of the thesis and outlines potential future directions. It is partially derived from:

  - **Chenhao Qu**, Rodrigo N. Calheiros, and Rajkumar Buyya, "A Taxonomy and Survey of Auto-scaling Web Applications in Clouds", *ACM Computing Surveys* (Under Review), 2016.

Chapter 2
Literature Review

This chapter aims to identify the open challenges and understand the strengths and limitations of the existing methods in auto-scaling and deployment of web applications in distributed computing Clouds through a thorough literature review. It covers the areas of Clouds service discovery, Clouds selection, and auto-scaling. We respectively survey existing proposals in these fields and compare them to identify their characteristics.

2.1 Introduction

APPLICATION providers have been more than ever dependent on the infrastructure of Cloud providers all over the world to host and manage their applications. In addition to the benefits brought by Cloud computing, utilizing resources from multiple Cloud data centers opens the following opportunities:

- **Better QoS:** Cloud providers can deploy their applications close to the end users that are geographically dispersed so that the network latencies experienced by distributed users are similarly acceptable.
- **Always-on Availability:** critical applications need to be available 24/7 without downtime. Deploying them in geographical dispersed data centers can ensure their survival even under disruptive disasters.
- **Regulation Compliance:** countries often limit application providers to store data of their citizens in locations under their sovereignty. Therefore, it is impossible to

This chapter is partially derived from: **Chenhao Qu**, Rodrigo N. Calheiros, and Rajkumar Buyya. "Auto-scaling Web Applications in Clouds: A Taxonomy and Survey", *ACM Computing Surveys*, 2016 (under review).

comply regulation requirements of different nations using a single data center.

- **Vendor Lock-in Avoidance:** it prevents the application providers only relying on one Cloud service and thus falling into vendor lock-in.
- **Cost-efficiency:** it allows the application providers to explore different offerings and pick the most cost-efficient one regarding their requirements.

Managing applications in multiple Clouds remain a challenging task. Appropriate techniques to select suitable services and provision resources need to be developed. Many researchers have been targeting these goals.

In this chapter, we aim to investigate, classify, and analyze the existing works that target these issues. Regarding each discussed issue, we first identify the related challenges, and then explain and compare the proposed works to understand their strengths and limitations to improve the state-of-the-art.

The rest of this chapter is organized as follows. Section 2.2 identifies challenges and reviews the developments in QoS-aware discovery of suitable Cloud services. Then we discuss the state-of-the-art of Clouds selection in Section 2.3. After that, we list the challenges of auto-scaling techniques for dynamically provisioning web applications and explain the existing approaches in Section 2.4. Finally, we summarize the findings in Section 2.5.

2.2 Clouds Service Discovery

Cloud service discovery is the procedure to find candidate Cloud services that can satisfy the application provider's requirements. It is an issue not unique to web applications. Most of the existing techniques are designed for general purpose usage of the Cloud. This process involves the following procedures:

- The client defines his functional and non-functional requirements about the Cloud services.
- The discovery service, according to the client's requirements, evaluate the public services.

- The discovery service returns a list (either ranked or unranked) to the client specifying services that it thinks can meet the user requirements.

Regarding web application, the task of service discovery is to identify providers that can offer satisfactory services for each component in the application regarding their individual functional and QoS requirements. The task can be achieved by respectively finding adequate services for each component using existing general purpose techniques and then intersecting the results to identify the service providers that can satisfy them all.

2.2.1 Challenges

The challenges involved in this process are summarized as follows:

Large Number of Services

Many providers operating their own Cloud services (as of 13 September 2016, CloudHarmony [84] lists 57 providers for computing services). Each provider owns several data centers and provides tens of VM service offerings. A large number of available services increases competition in the market and stimulates providers to provide better service with less cost. On the other hand, it also creates difficulty for Cloud users to evaluate all their choices to make the final decision efficiently.

Multiple Criteria

For each component or tier, application providers usually have multiple requirements regarding QoS and provided functions. For example, at the front-tier, an application provider might require the Cloud to offer firewall service and high network bandwidth for VM hosting the load balancer. At the second tier, the provider might focus more on CPU and memory performance. At the database tier, the throughput of disk I/O and security mechanisms are more important. Besides, often application associates various preferences to different requirements (e.g., CPU performance is more important than network performance). It is challenging to evaluate each Cloud service regarding multiple

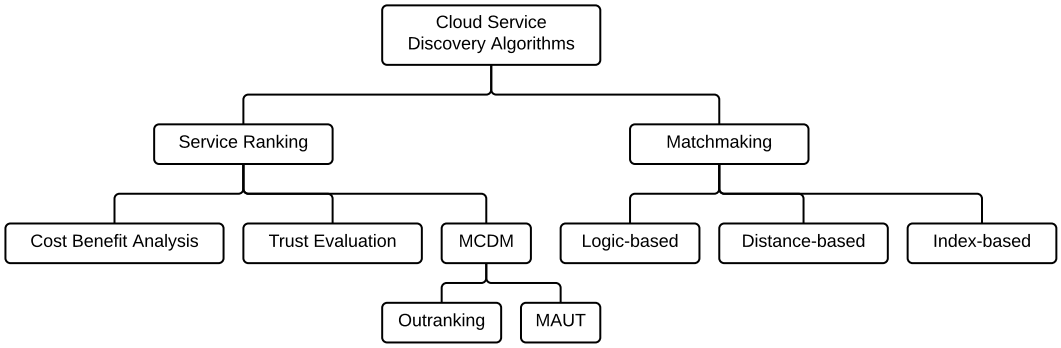


Figure 2.1: A taxonomy of Cloud service discovery algorithms

criteria and application provider's individual preferences of requirements.

Service Variation

Clouds are multi-tenant platforms, and thus their QoS is susceptible to workload changes and other factors [169]. For web applications, the performance stability and robustness of Cloud is important. Therefore, the discovery techniques that can quantify and compare services' stability according to their dynamic performances are more desirable.

2.2.2 State-of-the-art

Building a central repository with each Cloud services represented in a machine-understandable format is of vital importance to automatic Cloud services discovery. With this repository, the discovery engine can then automatically search satisfactory services according to Cloud client's requirements for each attribute he cares using various search algorithms. The ontology-based approach originated from managing semantic web services has been favored by the majority of the works, e.g., [51,111,130,178,189], because they can describe the properties of Cloud services in a machine understandable way and automatically match the semantics of Cloud clients' requirements. One working approach of this type is called Cloudy Metrics [178], which currently has a limited number of Cloud providers in its repository and only supports functional attributes.

The specific search algorithms adopted can be classified into two categories, namely service ranking and matchmaking. Figure 2.1 demonstrates our classification. Service

ranking approaches provide ranked lists of services according to their qualities. Based on how service is ranked, it can be further classified into cost-benefit analysis, trust evaluation and multi-criteria decision-making (MCDM) approaches. MCDM approaches can be divided into two sub-groups, namely multi-attribute utility theory (MAUT), which usually relies on utility functions to model an alternative's capability, and outranking, which tries to find domination relationships between alternatives through pairwise comparisons. Matchmaking approaches compare Cloud clients' requirements to actual service offerings to find satisfactory services. Some approaches in this category also provide ranking capability (e.g., distance-based approaches), while the others only support coarse-grained matching (e.g., logic-based approaches).

Service Ranking

Cost-benefit Analysis: Cost-benefit analysis approaches aim to find the IaaS service that can bring more profit and incur less risk for the business. These approaches should be able to estimate potential cost and benefit for each service and identify services that will incur an acceptable cost and promising benefits.

Calculating potential cost (including financial cost, reputation cost, security cost, etc.) of using Cloud involves building complex risk assessment models for different aspects at the Cloud clients side, especially when they have made legal commitments to their end users through SLA. Saripalli et al. [168] proposed a manual framework, called QUIRC, for organizations to assess the security risk on Clouds. Besides requiring human intervention, it ignores risk resilient abilities of different Cloud services. Djemame et al. [117] developed a promising risk assessment toolkit for OPTMIS project [64]. It uses risk inventory and historical database to automate assessment, management, and mitigation of the risks involved in the deployment.

To balance cost and benefit, Zeng et al. [221] calculated the simple weighted average of the total benefit and cost for using specific Cloud services. They are then ranked according to the results.

Trust Evaluation: Trust and reputation systems are commonly used for service discovery and selection for distributed systems [106]. The major advantage of them is that when applied to Cloud, their dynamic nature enables them to response to performance variations quickly.

Most commonly, trust systems use clients' ratings as the trust source. CloudHarmony [84] used to offer a platform for clients to rate the Cloud providers by giving a score from 0 to 5 like Amazon and eBays reputation systems. Noor and Sheng [150] proposed an improved system. It considers the credibility of the ratings when aggregating all the trust feedback.

The previous approaches ignore the fact that Clouds have multiple aspects. To overcome that, Habib et al. [80] designed an approach to evaluate the trustworthiness of Cloud services regarding QoS attributes defined in the Service Measurement Index (SMI) [48]. In their approach, each attribute in the SMI is rated from multiple sources, including user ratings, provider statements, measurements, and certificates [81]. Then they used CertainLogic [161] to combine trust values of different attributes to a single value.

Multi-criteria Decision Making (MCDM): MCDMs are systematic methodologies that rank alternatives A_1 to A_n regarding criteria C_1 to C_n with respectively associated weights W_1 to W_n. For each criterion, it is rated by the set of experts and aggregated using some algorithms. MCDM have been applied to solve ranking problems in many fields. When used for Cloud service discovery, the inputs are weights of each criterion assigned by the client. To facilitate automatic discovery, instead of using expert rating employed in other fields, the QoS value of each attribute usually come from dynamic performance measurements.

Analytical Hierarchy Process (AHP) is the most widely adopted MCDM method. Its goal is to evaluate the relative importance of attributes that are modeled in a hierarchical architecture, which well suits the attribute model of IaaS described in SMI [48]. At every level, the human evaluator is asked to assign a value, ranging from 1 to 9, to each pair of attributes regarding their relative importance. According to the pairwise scores, the weights of attributes at each level can be systematically determined. Then the alternatives

are evaluated by pairwise comparisons regarding each leaf attributes. Finally, the overall scores are aggregated from bottom-up by weighted summation. There are many works employ AHP to rank IaaS services. SMICloud [71] is the first attempt of AHP-based Cloud ranking. Other important works, such as [130, 142, 184], also adopt AHP or its variations.

Rehman et al. [198] compared seven different MCDM methods for ranking Cloud services, including five MAUT approaches, Min-Max, Max-Min, Compromise Programming, "Technique for Order Preference by Similarity to Ideal Solution" (TOPSIS), and AHP, and two outranking approaches, "ELimination and Choice Expressing Reality" (ELECTRE), and "Preference Ranking Organization Method for Enrichment Evaluations" (PROMEHEE). Their experimental results showed that different approaches lead to various decisions for the same sets of requirements, which indicates that the decision process itself has a significant impact on the final result. However, they did not come out with a judgment that which approach is better. To facilitate discovery in a dynamic environment, they further proposed an approach [199] considering the QoS variations in the Cloud. Its basic idea is to give more weight to the time measurements close to current time point instead of using a simple average so that the services perform better recently are more likely to be chosen.

Matchmaking

Logic-based: The basic idea of logic-based approaches is to use logic reasoning to determine whether the service is satisfactory to client's requirements or not. Dastjerdi et al. [51] proposed a matchmaking algorithm using description logic, in which user requirements and characteristics of services are modeled as sets. The matchmaking process involves calculating the overlapping degree of the two sets. Services are classified into Exact Match, PlugIn Match, Subsumption Match, Intersection Match, and No Match based on the overlapping degree. Liu et al. [130] used a hybrid approach that first identifies the functionally matched services using a reasoning algorithm and then ranks the services using MCDM according to their QoS. Cloudy Metrics [178] uses simple logic operators like $\geq$, $<$, AND, OR, etc. to search the satisfiable services. It only supports

functional attributes.

Distance-based: Distance-based approaches measure gaps between user requirements and service offerings and rank the services based on distances. Rehman et al. [197] proposed a MCDM approach using exponential weighted difference, in which services are evaluated as $e^{-(a_{i,1}-r_1)w_1} + e^{-(a_{i,2}-r_2)w_2} + ... + e^{-(a_{i,n}-r_n)w_n}$ where $a_{i,j}$ represents the ith provider's capability of jth attribute, and r_j and w_j respectively represents the user requirement and preference weight of the jth attribute. It can be viewed as a hybrid approach of MCDM and distance-based matchmaking.

Redl et al. [160] introduced an SLA matching approach for discovering Cloud services. In their model, the matchmaking is conducted between the client specified private SLA and provider defined public SLA. The matching has three phases. In the first step, they employed character-based string similarity metric Levenshtein distance and past cases to calculate the proximity of semantic element definitions (e.g., "Memory Consumption" vs. "Memory Usage"). Then it checks whether the metrics of definitions match. In the final stage, the probability of equality for every SLA element is evaluated by Support Vector Machines technique.

Amato et al. [8] developed a broker also using SLA matching. Their approach accepts client's requirements as hard constraints which the service must satisfy and soft constraints which are target values assigned by clients. For services matching all hard constraints, they are ranked by the aggregated utility calculated according to their distances to the soft constraints. In this process, clients can specify their preferences by defining different utility functions.

Index-based: Sundareswaran et al. [185] first attempted to index Cloud services to accelerate searching. They encoded attributes of Cloud services into search keys and stored them in a B+ tree based index. In this way, services with similar capabilities are stored closely and can be retrieved efficiently. When searching services, the k-nearest providers are first retrieved according to the key generated from user requirements. After that, the final ranking is conducted among the k candidates according to the client's preferences. Thanks to the indexing process, their algorithm is 100 times faster than exhaustive search

for a large number of services.

2.3 Clouds Selection

With the set of candidate Cloud services discovered, web application providers still face the problem of finally select a set of data centers offered by these services to deploy their applications according to various aspects. The factors considered by different applications when making the decisions are often diverse. Some may focus on application performance, some may pay more attention to cost, and some have to satisfy other constraints. In general, we summarize the challenges in this process as follows.

2.3.1 Challenges

Application Performance: One major motivation for using multiple distributed Cloud data centers to host web applications is to minimize the end user perceived latencies. Therefore, when selecting the hosting data centers, application providers should be aware of end user locations. Besides, the performance model of the application becomes more complicated when it has foreign dependencies to other services or applications.

In summary, with a set of users and their locations, finding a certain number of data centers from a set of candidates with minimum performance cost falls in the category of Facility Location Problems. They are NP-hard and notoriously difficult even to find good approximations [174].

Operational Cost: Cloud services can be expensive to applications. However, to accurately estimate the financial cost at the selection phase is difficult due to the dynamic nature of web application workloads. It is important to limit the number of selected data centers to keep the cost low, which involves finding the best balance between cost and performance.

Geographical Constraints: Besides performance issues, there are other limitations regarding locations of the selected data centers for some applications. The first limitation

comes from the fault-tolerant requirement. For some data-critical or availability-critical applications, they need their application replicas to be placed in geographically dispersed areas so that the data or application will survive even after catastrophic natural disasters. The second one is the legislation constraint. Data regulations in some countries require application providers to store the data of their citizens in particular domains.

Data Consistency: For applications utilizing multiple data centers, eventual consistency across data centers is acceptable in many cases. However, some applications, such as banking, and e-commerce applications, still need strong consistency even when data are replicated in different data centers. Complying strong consistency semantics across multiple data centers creates extra latency overhead, which cannot be ignored when selecting the data center locations.

Redeployment: Web application providers will face the needs to redeploy their applications in various situations. The most common case is that the application becomes increasingly attractive and the overall performance degrades under the current distribution of application instances. Other issues like pricing change, service level agreement (SLA) revision, and legislation problems may also trigger the web application providers to move their applications or deploy new instances.

Unfortunately, for most applications, migration to other data centers is costly and challenging, because of the associated data volume. Therefore, web application providers often tend to minimize such migrations. On the other hand, for some applications, efficient dynamic migration of data and services is plausible for applications with light data dependency, e.g., video transcoding and image rendering, and it can significantly improve the performance and save the providers considerable amount of cost.

2.3.2 State-of-the-art

Because of the various factors and assumptions considered by different application providers, it is hard to derive a clear classification of the existing research works in this field. Therefore, we list the individual characteristics of some important works in a table (Table 2.1)

Table 2.1: Characteristics of works related to Clouds selection.

Work	Environment	Goal	Algorithm
Li et al. [126]	CDN	minimize overall latency	dynamic programming
Qiu et al. [157]	CDN	minimize overall latency	heuristics for k-media
Benoit et al. [22]	CDN	minimize replication cost	heuristics
Tang et al. [190]	CDN	minimize network traffic	dynamic programming
Kang et al. [113]	Cloud	min latency/max coverage	local search/greedy
Kang et al. [112]	Cloud	minimize traffic of apps	iterative optimization
Zhang et al. [224]	Cloud	minimize cost for servers	model predictive control
Rodo. et al. [163]	Cloud	minimize cost for servers	divide and conquer
Ta et al. [188]	Cloud	min latency/bandwidth	heuristics
Ta et al. [187]	Cloud	minimize selected sites	greedy algorithms
Wu et al. [212]	Cloud	minimize total cost	model predictive control
He et al. [85]	Cloud	balance bandwidth & latency	a two-stage algorithm
Lei et al. [101, 102]	Cloud	data placement	meta-heuristic

and separately introduce them regarding their specific scenarios and solutions.

Before Cloud era, similar problems have been intensively studied in placing replicas into CDN network. They usually assume the network follows certain topologies [22, 126, 157] and then try to optimize the placements of objects regarding some objective functions, such as performance [126, 157], cost [22], and total network traffic [190]. Though these works are in a different context, they share commonalities with the problem of placing application replicas in Clouds.

Among them, Qiu et al.'s work [157] considered the closest optimization scenario to the Cloud context. They viewed each potential resource site to have unlimited capacity, which is an assumption commonly made in the Cloud context as well. Their objective is to minimize total network latency between customers and the corresponding closest replicas. They modeled it as a k-media problem. In Cloud context, Kang et al. [113] considered a similar scenario. They modeled the problem in two different approaches. Firstly, they also viewed it as a k-media problem. In the second method, they modeled it as a max k-cover problem, which aims to maximize the number of customers situate within the defined latency boundary of the closest application replica. They tested various heuristics proposed for the two problems, such as local search and greedy algorithm. Furthermore, they argued the placement of application replicas should be iteratively ad-

justed according to the change of customer distribution. However, they ignored the cost of migrating application replicas.

Kang et al. [112] investigated a scenario where Cloud applications are interdependent. Instead of just minimizing the traffic between applications and customers, they tried to minimize the total traffic including traffic exchanged between different applications. Their solution is to deploy the related applications cooperatively. They proposed an iterative optimization algorithm to solve the problem.

Zhang et al. [224] explored a holistic approach for server placement and auto-scaling. They assume that each potential data center has limited capacity and dynamic pricing. Their objective is to minimize the financial cost of the application provider by dynamically allocating servers across geographically dispersed data centers under the constraint of demand, capacity, and SLA. They employed the Model Predictive Control (MPC) framework to dynamically adjust the resource allocation in each data center and request routing from each user location. They further provided an analysis of the equilibrium outcome of their approach when multiple application providers use their strategy to scale their applications.

Rodolakis et al. [163] also investigated a holistic scenario. However, the difference is that they assume the capacity of each data center is unlimited. They aim to decide locations of servers, request routing of traffic, traffic routes in the network topology, and many servers and their types in each site so that the latency bound is met and cost of using resources is minimized. They decomposed the problem into three NP-hard subproblems: 1) find the route with the minimum cost that can satisfy the latency bound; 2) find the optimal number of servers and their types given the load; 3) solve the holistic optimization problem using the results from 1) and 2). Pseudo-polynomial and polynomial algorithms were respectively proposed for the three sub-problems.

Ta et al. [188] investigated how to place servers of Distributed Virtual Environment (DVE) applications across geographically dispersed Clouds. DVEs are a class of applications that customers interact with each other in a simulated 3D environment, which are highly sensitive to network latencies. DVEs are divided by zones, and a customer is tied to a particular zone unless he explicitly moves to another area, e.g., online gaming.

They also assume each server site has unlimited capacity. Their target is to select K sites among N ($K < N$) potential sites to place DVE servers so that the total network latencies or bandwidth consumption is minimized. They proposed several heuristic algorithms which aim to put the servers on some critical nodes of the network topology.

In another work, Ta et al. [187] introduced a new deployment setting. Different from their previous works, in this one, customers are associated with a contact server and a target server, which are not necessarily the same. Customers always send requests to their corresponding contact servers. If the target server and contact server are different, the request is forwarded to the target server. Their goal is to minimize the number of sites selected to satisfy the QoS requirements. They proposed two algorithms. One is based on a greedy algorithm, which iteratively selects the site that will result in the largest number of customers perceive acceptable QoS. The another algorithm operates in a similar greedy way, but it assumes all the contact and target server are the same for all the customers.

For some applications, the cost of serving and storing data counts a majority of the companies Cloud bills, e.g., video service and social network applications. When deployed in multiple Clouds, the placement of these applications is driven by the distribution of data.

Wu et al. [212] proposed a predictive method to dynamically optimize the placement and allocation of resources for social media streaming applications concerning operational cost across multiple Cloud data centers under a fluctuant workload. Operational cost includes storage, bandwidth, application VM renting, and video migration cost. Also in their optimization model, they wanted to guarantee the availability of each video, satisfy network latency bound, and meet bandwidth constraints. They solved the optimization problem by dual decomposition and the subgradient algorithm. To work online, they adopted a time look-ahead approach, which predicts the workload in next t time interval and then optimizes the resource allocation and data replication according to the estimated workloads.

He et al. [85] wanted to find the best balance between bandwidth cost and total network latency for serving video content to customers all over the world from geographically dispersed Clouds. They employed Nash Bargaining solution to ensure optimality

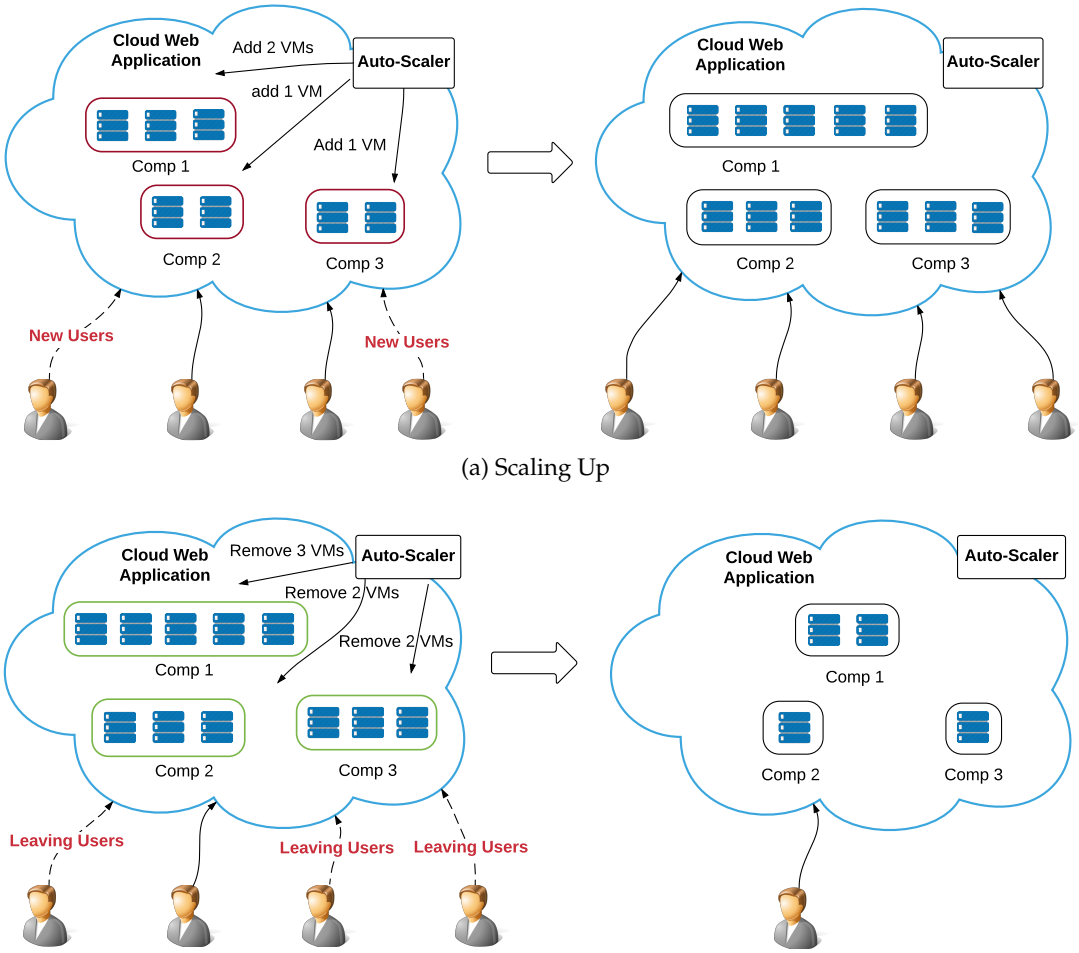
and fairness simultaneously. They used dual decomposition and subgradient algorithm to obtain the optimal bandwidth provisioning. Then they derived the video placement according to the bandwidth provisioning using a greedy algorithm.

Lei et al. [101, 102] studied the data placement problem across multiple Clouds in social network applications. In their setting, each user's data is associated with a master copy and a certain number of slave (read-only) copies. Their objective is to decide the placement of master and slave copies of each user in the social network graph so that some objective function is optimized. In [102], they assume all write requests are directed to the master copies and the read requests are relayed to other copies if the data are not stored in the contacting data center. Their objective function considers carbon footprint, operation distance (total geographical distance traveled by read and write requests), intra-Cloud traffic, and reconfiguration cost. They proposed a meta-heuristic algorithm to obtain approximate solutions. It iteratively optimizes the deployment of masters with the current placement of slaves and optimizes the deployment of slaves given the deployment of masters alternately. In [101], they considered a different placement strategy, in which they required a user's master copy must be collocated with all his friends' copies (either master or slave). They tried to minimize the storage, traffic, and redistribution cost under availability and QoS constraints. They proposed an algorithm that randomly selects users to swap their master and slave copies to find cost reduction opportunities online.

2.4 Auto-scaling

2.4.1 Challenges

The auto-scaling problem for web applications can be defined as how to autonomously and dynamically provision and deprovision a set of resources to cater for fluctuant application workloads so that resource cost is minimized and application service level agreements (SLAs) or service level objectives (SLOs) are satisfied. Figure 2.2 illustrates typical auto-scaling scenarios. In Figure 2.2a, due to increase in requests, the available resources are in congestion, and thus, the auto-scaler decides to provision certain resources respec-



(a) Scaling Up

(b) Scaling down

Figure 2.2: Typical auto-scaling scenarios

tively to each application component. Adversely, in Figure 2.2b, the auto-scaler deprovisions some resources from each component when the amount of requests has decreased.

This is a classic automatic control problem, which demands a system that dynamically tunes the type of resources and the amount of resources allocated to reach certain performance goals, reflected as the SLA. Specifically, it is commonly abstracted as a MAPE (Monitoring, Analysis, Planning, and Execution) control loop [116]. The control cycle continuously repeats itself as the time flows.

The biggest challenges of the problem lie in each phase of the loop as shown in Figure 2.3, which are listed and explained below.

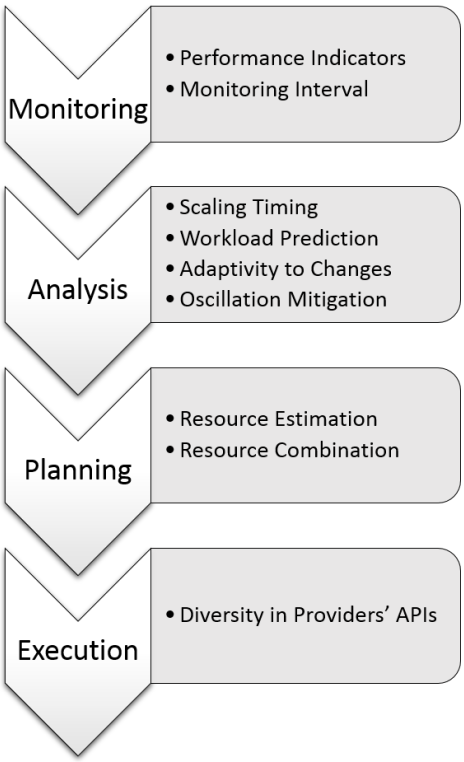


Figure 2.3: The challenges of auto-scaling web applications in each phase of the MAPE loop

Monitoring: The system needs to monitor some performance indicators to determine whether scaling operations are necessary and how they should be performed.

- Performance indicators: selection of the right performance indicators is vital to the success of an auto-scaler. The decision is often affected by many factors, such as application characteristics, monitoring cost, SLA, and the control algorithm itself.
- Monitoring interval: monitoring interval determines the sensitivity of an auto-scaler. However, very short monitoring intervals result in high monitoring cost both regarding computing resources and financial cost, and it is likely to cause oscillations in the auto-scaler. Therefore, it is important to tune this parameter to achieve balanced performance.

Analysis: During the analysis phase, the system determines whether it is necessary to perform scaling actions based on the monitored information.

- Scaling timing: the system firstly needs to decide when to perform the scaling actions. It can either proactively provision/deprovision resources ahead of the workload changes if they are predictable since the provision/deprovision process takes considerable time or it can perform actions reactively when workload change has already happened.

- Workload prediction: if the system chooses to scale the application proactively, how to accurately predict the future workload is a challenging task.

- Adaptivity to changes: sometimes the workload and the application may undergo substantial changes. The auto-scaler should be aware of the changes and timely adapt its model and settings to the new situation.

- Oscillation mitigation: scaling oscillation means the system frequently performs contradictory actions within a short period (i.e., acquiring resources and then releasing resources or vice versa). It should be prevented as it results in resource wastage and more SLA violations.

Planning: The planning phase estimates how many resources in total should be provisioned/deprovisioned in the next scaling action. It should also optimize the composition of resources to minimize financial cost.

- Resource estimation: the planning phase should be able to estimate how many resources are just enough to handle the current or incoming workload. This is a difficult task as auto-scaler has to figure out this information quickly without being able to execute the scaling plan to observe the real application performance, and it has to take the specific application deployment model into account in this process.

- Resource combination: to provision resources, auto-scaler can resort to both vertical scaling and horizontal scaling. If horizontal scaling is employed, as the Cloud providers offer various types of VMs, the system should determine to pick which of them for hosting the application. Another important factor is the pricing model of Cloud resources. Whether to utilize on-demand resources, reserved resources

or rebated resources greatly affects the total resource cost. All these factors form a huge optimization space, which is challenging to solve efficiently in short time.

Execution: The execution phase is responsible for actually executing the scaling plan to provision/deprovision the resources. It is straightforward and can be implemented by calling Cloud providers' APIs. However, from an engineering point of view, being able to support APIs of different providers is a challenging task.

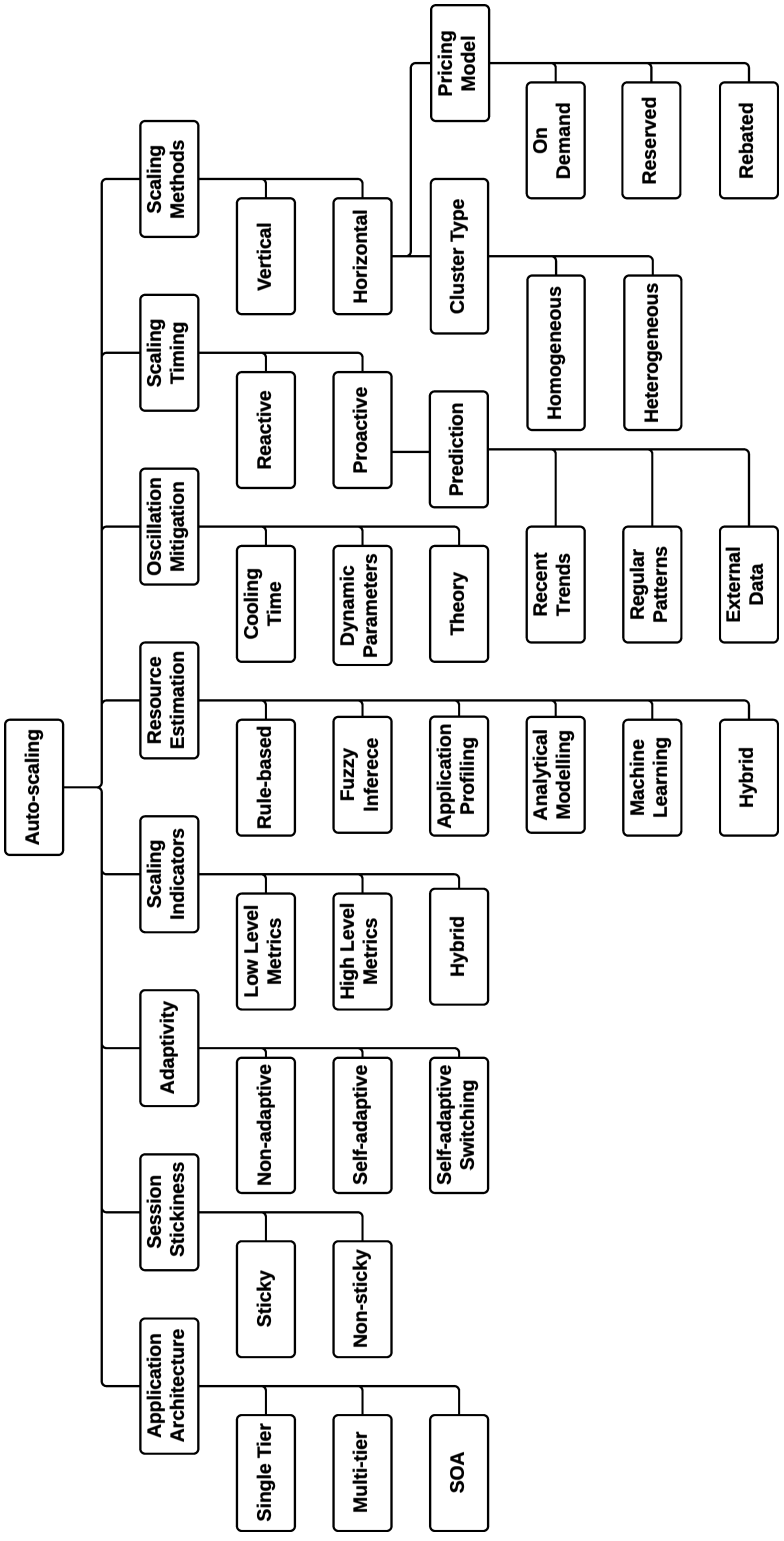


Figure 2.4: The taxonomy for auto-scaling web applications in Clouds

2.4.2 Taxonomy

Figure 2.4 illustrates our proposed taxonomy for auto-scaling web applications. It classifies the existing works based on the identified challenges in each of the MAPE phase in Section 2.4.1 and their targeted environment. Particularly, the taxonomy covers the following aspects in auto-scaling:

- Application Architecture: the architecture of the web application that the auto-scaler is managing.
- Session Stickiness: whether the auto-scaler supports sticky session.
- Adaptivity: whether and how the auto-scaler adjusts itself to adapt to workload and application changes.
- Scaling Indicators: what metrics are monitored and measured to make scaling decisions.
- Resource Estimation: how the auto-scaler estimates the amount of resources needed to handle the workload.
- Oscillation Mitigation: how the auto-scaler reduces the chance of provision oscillation.
- Scaling Timing: whether the auto-scaler supports proactively scaling the application and how it predicts future workload.
- Scaling Methods: how the auto-scaler decides what methods to use to provision resources and what combination of resources are provisioned to the application.

Existing approach span across different subcategories and are discussed in each of them (i.e., an auto-scaler is built for multi-tier applications, and employs proactive scaling with machine learning resource estimation techniques). Note that this taxonomy is based on features and thus does not reflect the relative performance of the discussed approaches. Because the surveyed works target diverse workload patterns, application architectures, and pricing models, there is no single answer to the question that which

approach performs the best. Based on the taxonomy and explanation of the concepts, we list characteristics of the surveyed works in Table 2.2. In the following Sections (from Section 2.4.3 to Section 2.4.15), the existing auto-scalers are introduced and compared according to this taxonomy.

2.4.3 Application Architecture

There are three types of web application architectures mentioned in the literature: namely single tier, multi-tier, and service-oriented architecture.

Single Tier/Single Service

A tier represents the software function implemented and packaged as the minimum interactive module in a layered software stack. In a production deployment, a server usually exclusively host a single software tier, and within a tier, a load balancer is used to balance and dispatch load among the participating instances of the tier cluster. Single tier architecture by definition is the architecture in which application is composed of only one tier. Relatively, the architecture with multiple connected software tiers is called multi-tier architecture. Instead of calling single tier as an application architecture, it is more accurate to think it as the smallest granularity that can be possibly managed by an auto-scaler, since hardly any web application is composed of only one tier.

Nowadays web applications are becoming more and more complicated and deviate from the traditional multi-tier architecture. In those cases, the fundamental scaling component is often referred as a service. The majority of the existing auto-scalers separately manage each single tier or service within an application instead of considering it as a whole. This method is both simple and general. However, it often results in globally suboptimal resource provision as it requires to divide the SLA requirements of the overall application into sub-requirements of each tier or service, which is often a challenging and subjective task.

Table 2.2: A Review of auto-scaling properties of key works for single Cloud

Work	Application Architecture	Sticky Session	Adaptivity	Scaling Indicators	Resource Estimation	Oscillation Mitigation	Proactive	Scaling Methods
Dolye et al. [55]	single-tier	✓	non-adaptive	hybrid	analytical model	—	×	vertical
Kamra et al. [110]	3-tier	✓	self-adaptive	high-level	analytical model	—	×	vertical
Tesauro [191]	single-tier	×	self-adaptive	high-level	reinforcement learning	—	×	hom. horizontal
Tesauro et al. [192]	single-tier	×	self-adaptive	high-level	hybrid	—	×	hom. horizontal
Jing et al. [103]	single-tier	✓	self-adaptive	high-level	hybrid	—	×	vertical
Villela et al. [206]	single-tier	×	non-adaptive	hybrid	analytical model	—	×	hom. horizontal
Zhang et al. [223]	multi-tier	—	—	hybrid	hybrid	—	—	—
Chen et al. [37]	single-tier	✓	self-adaptive	hybrid	regression	—	✓	hom. horizontal
Urgaonkar et al. [202]	multi-tier	×	non-adaptive	high-level	analytical model	—	✓	hybrid
Iqbal et al. [91]	single-tier	×	non-adaptive	high-level	rule-based	—	×	hetr. horizontal
Lim et al. [129]	single-tier	×	self-adaptive	low-level	rule-based	dynamic para.	×	hom. horizontal
Bodik et al. [26]	single-tier	×	self-adaptive	high-level	regression	dynamic para.	✓	hom. horizontal
Padala et al. [152]	multi-tier	✓	self-adaptive	hybrid	regression	dynamic para.	×	vertical
Kalyvianaki et al. [109]	single-tier	✓	self-adaptive	low-level	rule-based	—	×	vertical
Lama and Zhou [123]	multi-tier	×	self-adaptive	high-level	fuzzy inference	dynamic para.	×	hom. horizontal
Lim et al. [128]	storage-tier	×	self-adaptive	low-level	rule-based	dynamic para.	×	hom. horizontal
Dutreilh et al. [56]	single-tier	×	self-adaptive	high-level	hybrid	cooling time	×	hom. horizontal
Gong et al. [76]	single-tier	✓	self-adaptive	low-level	hybrid	—	✓	vertical
Islam et al. [93]	single-tier	×	self-adaptive	low-level	neural net./regression	—	✓	—
Lama and Zhou [122]	multi-tier	×	self-adaptive	high-level	hybrid	—	×	hom. horizontal
Bi et al. [23]	multi-tier	×	non-adaptive	high-level	analytical model	—	×	hom. horizontal
Singh et al. [177]	multi-tier	×	non-adaptive	high-level	analytical model	—	×	hom. horizontal
Jiang et al. [98]	SOA	×	self-adaptive	high-level	analytical model	—	×	hom. horizontal
Chieu et al. [39]	single-tier	✓	non-adaptive	high-level	rule-based	—	×	hom. horizontal

Continued on next page

Table 2.2 – continued from previous page

Work	Application Architecture	Sticky Session	Adaptivity	Scaling Indicators	Resource Estimation	Oscillation Mitigation	Proactive	Scaling Methods
Dutreilh et al. [57]	single-tier	✗	self-adaptive	high-level	reinforcement learning	—	✗	hom. horizontal
Li and Venugopal [127]	single-tier	✗	self-adaptive	low-level	reinforcement learning	—	✗	hom. horizontal
Caron et al. [32]	single-tier	✗	self-adaptive	low-level	string matching	—	✓	—
Huber et al. [89]	single-tier	✗	non-adaptive	hybrid	rule-based	—	✗	hybrid
Iqbal et al. [92]	multi-tier	✗	self-adaptive	hybrid	hybrid	—	✗	hom. horizontal
Jiang et al. [99]	multi-tier	✗	non-adaptive	hybrid	online profiling	—	✗	hetr. horizontal
Malkowski et al. [137]	multi-tier	✗	self-adaptive	hybrid	hybrid	—	✗	hom. horizontal
Roy et al. [164]	multi-tier	✗	non-adaptive	hybrid	analytical model	—	✓	hom. horizontal
Upendra et al. [200]	multi-tier	✗	non-adaptive	high-level	profiling	—	✓	hetr. horizontal
Vasic et al. [205]	single-tier	✗	self-adaptive	low-level	online profiling	—	✗	hom. horizontal
Ali-Eldin et al. [4]	single-tier	✗	self-adaptive	high-level	analytical model	—	✓	hom. horizontal
Dawoud et al. [52]	single-tier	✗	non-adaptive	low-level	rule-based	—	✗	compare ver. hor.
Fang et al. [62]	single-tier	✗	—	—	—	—	✓	—
Yazdanov et al. [217]	single-tier	✓	self-adaptive	low-level	regression	—	✓	vertical
Ghanbari et al. [74]	single-tier	✗	non-adaptive	high-level	analytical model	—	✗	hetr. horizontal
Zhu and Agrawal [228]	single-tier	✓	self-adaptive	low-level	reinforcement learning	—	✗	vertical
Dutta et al. [58]	multi-tier	✗	non-adaptive	hybrid	application profiling	—	✗	hybrid
Gandhi et al. [70]	multi-tier	✗	non-adaptive	hybrid	profiling	—	✗	hom. horizontal
Rui et al. [165]	multi-tier	✗	non-adaptive	high-level	rule-based	—	✗	hybrid
Jiang et al. [100]	single-tier	✗	non-adaptive	high-level	analytical model	—	✓	hom. horizontal
Al-Haidari et al. [1]	single-tier	✗	non-adaptive	high level	rule-based	—	✗	hom. horizontal
Bu et al. [27]	single-tier	✓	self-adaptive	high level	reinforcement learning	—	✗	vertical
Gambi et al. [67]	single-tier	✗	self-adaptive	low-level	Kriging regression	—	✗	hom. horizontal
Barrett et al. [21]	single-tier	✗	self-adaptive	high-level	reinforcement learning	—	✗	hetr. horizontal
Sedaghat et al. [170]	single-tier	✗	non-adaptive	high-level	—	—	✗	hetr. horizontal

Continued on next page

Table 2.2 – continued from previous page

Work	Application Architecture	Sticky Session	Adaptivity	Scaling Indicators	Resource Estimation	Oscillation Mitigation	Proactive	Scaling Methods
Yazdanov et al. [216]	single-tier	✗	self-adaptive	hybrid	reinforcement learning	—	✓	vertical
Ali-Eldin et al. [5]	single-tier	✗	switch	—	—	—	✓	hom. horizontal
Almeida Morais et al. [6]	single-tier	✗	self-adaptive	low-level	various regressions	—	✓	hom. horizontal
Nguyen et al. [146]	multi-tier	✗	non-adaptive	hybrid	online profiling	—	✓	hom. horizontal
Herbst et al. [87]	—	✗	self-adaptive	—	—	—	✓	—
Grozev and Buyya [78]	single-tier	✓	non-adaptive	low-level	rule-based	—	✗	hom. horizontal
da Silva Dias et al. [50]	single-tier	✗	non-adaptive	hybrid	rule-based	—	✓	hom. horizontal
Loff and Garcia [132]	single-tier	✗	non-adaptive	low-level	rule-based	—	✓	hom. horizontal
Cunha et al. [49]	single-tier	✗	self-adaptive	low-level	rule-based	theory	✗	hom. horizontal
Netto et al. [145]	single-tier	✗	self-adaptive	low-level	rule-based	theory	✗	hom. horizontal
Aniello et al. [16]	single-tier	✗	non-adaptive	high-level	analytical model	—	✓	hom. horizontal
Frey et al. [65]	single-tier	✗	non-adaptive	hybrid	fuzzy inference	—	✓	hom. horizontal
Yang et al. [214]	single-tier	✗	non-adaptive	hybrid	rule-based	—	✓	hetr. horizontal
Fernandez et al. [63]	single-tier	✗	non-adaptive	high-level	profiling	—	✗	hetr. horizontal
Srirama and Ostovar[182]	single-tier	✗	non-adaptive	—	—	—	✗	hetr. horizontal
Gandhi et al. [68]	single-tier	✗	self-adaptive	high-level	analytical model	—	✗	hybrid
Spinner et al. [181]	single-tier	✓	self-adaptive	hybrid	analytical model	—	✗	vertical
Gergin et al. [72]	multi-tier	✗	non-adaptive	high-level	analytical model	—	✗	hom. horizontal
Han et al. [82]	multi-tier	✗	non-adaptive	high-level	analytical model	—	✗	hom. horizontal
Kaur and Chana [114]	multi-tier	✗	non-adaptive	high-level	analytical model	—	✓	hom. horizontal
Gandhi et al. [69]	multi-tier	✗	self-adaptive	hybrid	analytical model	—	✗	hom. horizontal
Nikravesh et al. [147]	—	—	—	—	—	—	✓	—
Yanggratoke et al. [215]	single-tier	✗	self-adaptive	high-level	batch & online learning	—	✗	hom. horizontal
Grimaldi et al. [77]	single-tier	✗	self-adaptive	low-level	rule-based	—	✗	hom. horizontal
Gambi et al. [66]	single-tier	✗	self-adaptive	high-level	hybrid	—	✗	hom. horizontal

Continued on next page

Table 2.2 – continued from previous page

Work	Application Architecture	Sticky Session	Adaptivity	Scaling Indicators	Resource Estimation	Oscillation Mitigation	Proactive	Scaling Methods
Salah et al. [166]	single-tier	×	non-adaptive	high-level	analytical model	—	×	hom. horizontal
Iqbal et al. [90]	multi-tier	×	self-adaptive	high-level	reinforcement learning	—	×	hom. horizontal
Amazon [11]	single-tier	×	non-adaptive	high/low	rule-based	cooling time	×	hom. horizontal
RightScale [162]	single-tier	×	non-adaptive	high/low	rule-based	cooling time	×	hom. horizontal
Qu et al. [158]	single-tier	×	non-adaptive	low-level	profiling	—	×	hetr. horizontal
Jamshidi et al. [95]	single-tier	×	self-adaptive	high-level	hybrid	—	×	hom. horizontal
Grozev and Buyya [79]	single-tier	×	self-adaptive	hybrid	rule-based	—	×	hetr. horizontal

Multi-tier

Multi-tier applications, as introduced in the previous section, are composed of sequentially connected tiers. At each tier, the request either relies on the downstream tier to complete its processing or it is returned to the upstream tier and finally to the user.

A widely-adopted architecture of this type usually consists three tiers: one frontend, one application logic, and one database tier. The database tier is often considered dynamically unscalable and ignored by the auto-scalers.

Many works have targeted multi-tier applications. Some of them employ the divide and conquer approach that breaks overall SLA into SLA of each tier, such as the works conducted by Urgaonkar et al. [202], Singh et al. [177], Iqbal et al. [92], Malkowski et al. [137], Upendra et al. [200], and Gergin et al. [72]. Others consider SLA of the whole application and provision the resources to each tier holistically. This strategy requires more efforts in modeling and estimating resource consumption using sophisticated queuing networks and machine learning techniques as discussed in Section 2.4.11, and the resulted auto-scalers are only applicable to multi-tier applications. Important works of this kind include approaches proposed by Zhang et al. [223], Jung et al. [108], Padala et al. [152], Lama and Zhou [122, 123], Han et al. [82], and Kaur and Chana [114].

Service-Oriented Architecture (Microservices)

Service-oriented architecture (SOA) or microservice architecture has now become the dominant paradigm for large web applications, such as Amazon e-commerce website, and Facebook. In this kind of architecture, applications are composed of standalone services that interact with each other through pre-defined APIs. More importantly, the services are not necessarily connected sequentially as in multi-tier applications. SOA applications are commonly abstracted as directed graphs with each nodes representing services and directed edges as their interactions.

Due to its complexity, it is hard to manage resource provision of all the services holistically. Therefore, industry and most works employ the divide and conquer approach. Differently, Jiang et al. [98] proposed a method that can satisfy SLA of the whole SOA

application. It is based on a bottom-up approach with each service estimating its performance after having one instance added or removed. Then it determines that scaling which service can bring the greatest benefit regarding response time.

2.4.4 Session Stickiness

A session is a series interactions between a client and the application. After each operation, the client halts to read its feedback given by the application and then issues the next move. To ensure a seamless experience, it is necessary to keep the intermediate statuses of the clients during their sessions. Otherwise, the operations conducted by the clients will be lost and they have to repeat the previous operations to proceed. Taking a social network application as an example, a session can involve the following operations: the client first accesses the home page and then logs into the application; after that, he performs several actions such as viewing his and his friends' timeline, uploading photos, and updating his status, before he quits the application.

This session-based access pattern has caused issues on efficiently utilizing elastic resources in Cloud because the stateful nature of session forces the user to be connected to the same server each time he submits a request within the session if the session data is stored in the server. Such sessions are considered sticky. They limit the ability of the auto-scaler to terminate under-utilized instances when there are still unfinished sessions handled by them. Therefore, it is regarded a prerequisite to transforming stateful servers into stateless servers before an auto-scaler can manage them.

There are multiple ways to achieve this, and a complete introduction to them is out of the scope of this chapter. The most adopted approach is to move the session data out of the web servers and store them either at user side or in a shared Memcached cluster.

Though most auto-scalers require the scaling cluster to be stateless, there do exist exception auto-scalers that can handle stateful instances. Chieu et al. [39, 40] proposed an auto-scaler based on the number of active sessions in each server. They restricted a server can be terminated only when there is no active session in it. Grozev and Buyya [78] proposed a better approach by integrating a similar auto-scaler with a load balancing algorithm that consolidates sessions into as few instances as possible.

2.4.5 Adaptivity

Auto-scalers fall in the realm of control systems. As stated in the introduction, they involve tuning the resources provisioned to the application to reach the target performance. One major issue coupled with the design of a control system is its adaptivity to changes. As in dynamic production environment, workload characteristic, and even the application itself can change at any moment. Therefore, adaptivity is important to auto-scalers. Based on the level of adaptivity, we classify the existing works into three categories.

2.4.6 Non-adaptive

In the non-adaptive approaches, the control model is predefined, and they make decisions purely based on the current input. Examples are the rule-based approaches employed by the industry, such as Amazon Auto-Scaling service [11]. They require the user to define a set of scaling up and scaling down conditions and actions offline. During production time, the auto-scaler makes scaling decisions only when the conditions are met. They do not allow automatic adjustment of the settings during production. When using this type of auto-scalers, the users often need to spend reasonable effort in offline testing to find the proper configuration.

2.4.7 Self-adaptive

Self-adaptive auto-scalers are superior to their non-adaptive counterparts. Though the core control models in them are fixed as well, they are capable of autonomously tune themselves according to the real-time quality of the control actions observed. In this way, the designer only needs to determine the core control model, such as whether it is linear or quadratic, and the auto-scaler will adjust and evolve itself to meet the target performance. This feature can be implemented through extending the pre-existing self-adaptive control frameworks in control theory, such as Kamra et al. [110], Kalyvianaki et al. [109], and Grimaldi et al. [77]'s work. Self-adaptivity can also be realized through dynamic measurement or correction of parameters in analytical models and machine learning approaches, such as reinforcement learning and regression. The detailed

explanations of them are given in Section 2.4.11.

The benefit of introducing self-adaptivity is that it significantly reduces the amount of offline preparation required to utilize an auto-scaler. Furthermore, once substantial changes are detected, self-adaptive approaches can autonomously abort the current model and retrain itself, thus, mitigating the maintenance effort as well. Their primary drawback is that it usually takes time for them to converge to a good model and the application will suffer from bad performance during the early stage of training.

Self-adaptive Switching

Beyond utilizing a single self-adaptive module, some auto-scalers have employed a more adaptive framework, which we call self-adaptive switching. In these approaches, they simultaneously apply multiple non-adaptive or self-adaptive controllers and actively switch control between controllers based on their observed performance on the application. The included self-adaptive controllers continuously tune themselves in parallel. However, at each moment, only the selected best controller can provision resources. Patikirikorala et al. [153] employed this approach and Ali-Eldin et al. [5] proposed a self-adaptive switching approach based on the classification of the application workload characteristics, i.e., their periodicity and the burstiness.

2.4.8 Scaling Indicators

The actions of auto-scalers are based on performance indicators of the application obtained through the monitoring phase. These indicators are produced and monitored at different levels of the system hierarchy from low-level metrics at the physical/hypervisor level to high-level metrics at the application level.

Low-Level Metrics

Low-level metrics, in the context of this chapter, are server information monitored at the physical server/virtual machine layer by hypervisors, such as utilization of CPU, memory, and network resources, memory swap, and cache miss rate. These data can be ob-

tained through monitoring platform of the Cloud provider or from monitoring tools for operating systems. However, it is a non-trivial task to accurately infer the observed application performance merely according to the low-level metrics, and therefore, makes it a difficult task to make sure that the SLA can be met faithfully with the available resources.

Designing an auto-scaler which solely monitors low-level performance indicators is possible. The simplest solution is to use the utilization of CPU and other physical resources as indicators and scale up and scale down resources to maintain the overall utilization within a predefined upper and lower bound. Industry systems widely adopt this approach.

2.4.9 High-Level Metrics

High-level metrics are performance indicators observed at the application layer. Those useful to auto-scaling include resource rate, average response time, session creation rate, throughput, service time, and request mix.

Some metrics, like request rate, average response time, throughput, and session creation rate, are easy to measure. They alone enable operation of an auto-scaler. The easiest method to construct one is to replace utilization metrics in the simple auto-scaler mentioned in the previous section with any of such high-level metric. However, auto-scalers employing this approach are not able to accurately estimate the amount of resources needed and often over or under provision resources.

Some approaches require obtaining the information about request service time and request mix [114,177,223] to estimate how much resources needed. These metrics are not straightforward to measure.

Service time is the time a server spent on processing the request, which is widely used in the queuing models to approximate the average response time or sojourn time. Except for a few works [72,82] that assume this metric as known a priori, to accurately measure it, either offline profiling [156] or support from the application [16] is required. Therefore, instead of directly probing it, some works use other approaches to approximate it. Kaur and Chana [114] mentioned the use of past server logs to infer the mean service time. Gandhi et al. [69] employed Kalman filters to estimate service time during runtime.

Zhang et al. [223] used a regression method to make the approximation. Jiang et al. [99] resorted to profiling each server when it is first online using a small workload without concurrency and then estimating service time through queuing theory. In another work, Jiang et al. [98] utilized a feedback control loop to adjust the estimation of service time at runtime.

Request mix is hard to measure because an understanding of the application is essential to distinguish different types of requests. Designing a mechanism to accurately classify various types of requests from outside of the application itself is an interesting and challenging problem to be explored.

2.4.10 Hybrid Metrics

In some auto-scalers, both high-level and low-level metrics are monitored. A common combination is to observe request rate, response time, and utilization of resources. Some auto-scalers [69,156] monitor them because the queueing models employed for resource estimation require them as input. Some works [58,63,103,152,216] use these hybrid metrics to dynamically build a model relating specific application performance to physical resource usage through online profiling, statistical, and machine learning approaches, thus increasing the accuracy of resource estimation without constructing complex analytical models. Another important reason to monitor request rate along with low-level metrics is to conduct future workload prediction [58,164].

Besides low-level and high-level metrics from the platform and application, other factors outside may also play a significant role. For example, Frey et al. [65], in their fuzzy-based approach, utilize other related data, such as weather, and political events to predict workload intensity.

2.4.11 Resource Estimation

Resource estimation lies in the core of auto-scaling as it determines the efficiency of resource provisioning. It aims to identify the minimum amount of computing resources required to process the workload to determine whether and how to perform scaling op-

erations. Accurate resource estimation allows the auto-scaler to quickly converge to the optimal resource provision, while estimation errors either result in an insufficient provision, which leads to inevitable delay of the provisioning process and increased SLA violations, or resource wastage that incurs more cost.

Various attempts have been made to develop resource estimation models from basic approaches to methods with sophisticated models. We categorize them into six groups, namely rule-based, fuzzy inference, application profiling, analytical modeling, machine learning, and hybrid approaches. The existing methods in each cluster are explained and compared afterward.

Rule-based Approaches

Rule-based approaches are widely adopted by industry auto-scalers, such as Amazon Auto-Scaling Service [11]. Its kernel is a set of predefined rules consisting of triggering conditions and corresponding actions, such as "If CPU utilization reaches 70%, add two instances", and "If CPU utilization decreases below 40%, remove one instance". As stated in Section 2.4.8, users can use any metrics, low-level or high-level, to define the triggering conditions, and the control target of the auto-scaler is usually to maintain the concerned parameters within the predefined upper and lower threshold. Theoretically, the simple rule-based approach involves no accurate resource estimation; only empirical guessing hard coded in the action part of the rule as adding or removing certain amount or percentage of instances. As the simplest version of auto-scaling, it commonly serves as benchmark for comparison and is used as the basic scaling framework for works that focus on other aspects of auto-scaling, such as Dawoud et al.'s work [52] which aims to compare vertical scaling and horizontal scaling, and Rui et al.'s work [165] which considers all possible scaling methods, or prototyping works, like the one carried out by Iqbal et al. [91].

Though simple rule-based auto-scaler is easy to implement, it has two significant drawbacks. The first is that it requires an understanding of the application characteristics and expert knowledge to determine the thresholds and proper actions. Al-Haidari et al. [1] conducted a study to show that these parameters significantly affect auto-scaler's

performance. The second is that it cannot adapt itself when dynamic changes occur to workload and application.

Hard coded number of instances to scale up and scale down, called step sizes, becomes inappropriate when the workload changes dramatically. For example, if the application is provisioned by four instances at the start, adding one instance will boost 25% of the capability. After a while, the cluster has increased to ten instances due to workload surge, adding one instance in this case only increases 10% of capacity. Improvements are made to the basic model using adaptive step sizes. Netto et al. [145] proposed an approach that decides the step size holistically at runtime based on the upper threshold, the lower threshold, and the current system utilization. It first deduces the upper and lower bounds respectively for step sizes of scaling up and scaling down operations to prevent oscillation and then scale the step sizes using a fixed parameter representing aggressiveness of the auto-scaler determined by the user. They reported the adaptive strategy performed best for bursty and peaky workload but lead to limited improvements for other types of workloads. Cunha et al. [49] employed a similar approach. However, in their approach, the aggressiveness parameter is also dynamically tunable according to QoS requirements.

In addition to the step size, fixed thresholds also could cause inefficient resource utilization. For instance, the thresholds of 70% and 40% may be suitable for a small number of instances but are inefficient for large clusters as single instance has a subtle impact on the overall utilization and a lot of instances actually can be removed before the overall usage reaching the 40% lower bound. A solution to mitigate this problem is also to make the thresholds dynamic. Lim et al. [128, 129] used this approach.

RightScale [162] proposes another important variation of the simple rule-based approach. Its core idea is to let each instance decide whether to shrink or expand the cluster according to predefined rules and then utilize a majority voting approach to make the final decision. Calcavecchia et al. [29] also proposed a decentralized rule-based auto-scaler. In their approach, instances are connected as a P2P network. Each instance contacts its neighbors for their statuses and decides whether to remove itself or start a new instance in a particular probability derived from their statuses.

Fuzzy Inference

Fuzzy-based auto-scalers can be considered as advanced rule-based auto-scalers as they rely on fuzzy inference, the core of which is a set of pre-defined If-Else rules, to make provision decisions. The major advantage of fuzzy inference compared to simple rule-based reasoning is that it allows users to use linguistic terms like "high, medium, low", instead of accurate numbers to define the conditions and actions, which makes it easier for human beings to effectively represent their knowledge (human expertise) about the target. Fuzzy inference works as follows: the inputs are first fuzzified using defined membership functions; then the fuzzified inputs are used to trigger the action parts in all the rules in parallel; the results of the rules are then combined and finally defuzzified as the output for control decisions. Representative approaches of this kind include the one proposed by Frey et al. [65] and the work conducted by Lama and Zhou [123]. Due to the complexity of manually designing the rule set and possible changes happening during runtime, fuzzy-based auto-scalers are commonly coupled with machine learning techniques to automatically and dynamically learn the rule set [95,103,122]. Their details are introduced in Section 2.4.11.

Application Profiling

We define profiling as a process to test the saturating point of resources when running the specific application using synthetic or recorded real workload. Application profiling is the simplest way to accurately acquire the knowledge of how many resources are just enough to handle the given amount of workload concurrently. Tests need to be conducted either offline or on the fly to profile an application.

Offline profiling can produce the complete spectrum of resource consumption under different levels of workload. With the obtained model, the auto-scaler can more precisely supervise the resource provisioning process. Upendra et al. [200], Gandhi et al. [70], Fernandez et al. [63], and Qu et al. [158] employed this approach. The drawback of this approach is that the profiling needs to be reconducted manually every time the application is updated.

Profiling can be carried out online to overcome this issue. However, the online environment prohibits the auto-scaler to fine-grainedly profile the application as a VM should be put into service as soon as possible to cater the increasing workload. Vasić et al. [205] proposed an approach that first profiles the application, then classifies the application signatures into different workload classes (number of machines needed). When changes happen to the application, the profiled new application characteristics are fed into the trained decision tree to carry out quick resource provisioning by finding the closest resource allocation plan stored before. Nguyen et al. [146] relied on online profiling to derive a resource estimation model for each application tier. When profiling each tier, other tiers are provisioned with ample resources. In this way, one by one, models for all the tiers are obtained. Jiang et al. [99] proposed a quick online profiling technique for multi-tier applications by studying the correlation of resource requirements that different tiers pose on the same type of VM and the profile of a particular tier on that type of VM. This approach allows them to roughly deduce performance of the VM on each tier without actually running each tier on it. Thus, the newly acquired VM can be put into service in relatively quicker speed.

Analytical Modeling

Analytical modeling is a process of constructing mathematical models based on theory and analysis. For resource estimation problems in auto-scaling, dominant models are built upon queuing theory [75].

In the generalized form, a queue can be represented as $A/S/C$, where A is the distribution of time interval between arrivals to the queue, S is the distribution of time required to process the job, and C stands for the number of servers. Common choices for A in the existing works are M (Markov) which means that arrivals follow the Poisson process, and G (General) which stands the inter-arrival time has a general distribution. For S, the prominent alternatives are M (Markov) which represents exponentially distributed service time, D (Deterministic) which means the service time is fixed, and G (General) which stands the service time has a general distribution. Detailed introduction of different types of queues is out of the scope of this chapter.

For a single application, tier, or service, if the underlying servers are homogeneous, it is more convenient to abstract the whole application/tier/service as a single queue with one server. Kamra et al. [110], Villela et al. [206], Gandhi et al. [68, 69], and Gergin et al. [72] employed this method. Some described the cluster using a queue with multiple servers, like Ali-Eldin et al. [4], Jiang et al. [100], Aniello et al. [16], and Han et al. [82]. Other works modeled each server as a separate queue, such as the ones proposed by Doyle et al. [55], Urgaonkar et al. [202], Roy et al. [164], Ghanbari et al. [74], Kaur and Chana [114], Spinner et al. [181], and Jiang et al. [98]. Bi et al. [23] proposed a hybrid model, in which the first tier is modeled as an M/M/c queue while other tiers are modeled as M/M/1 queues. Different from the traditional queuing theory, Salah et al. [166] used an embedded Markov chain method to model the queuing system.

When the application involves multiple tiers or is composed of many services, single layer queuing models are insufficient. Instead, a network of queues is needed to describe the components and their relations. These models are known as queuing networks. As introduced in Section 2.4.3, to decide a number of resources in each component, there are two strategies. One is to divide the SLA into separate time portions and distribute them to each component. By this method, the queuing model for each component can be easily solved. However, it usually results in suboptimal solutions globally. Another method is to holistically provision resources to all the components to satisfy the SLA. Such method is more challenging as it is difficult and computationally heavy to find the optimal resource provision plan regarding a complex queuing network model.

Some models and methods have been proposed to tackle the challenge. Villela et al. [206] described the model as an optimization problem and used three different approximations to simplify it. Bi et al. [23] as well employed an optimization approach. Roy et al. [164] and Zhang et al. [223] utilized MVA (Mean Value Analysis), a widely adopted technique for computing expected queue lengths, waiting time at queuing nodes, and throughput in equilibrium for a closed queuing network, to anticipate the utilization at each tier under the particular provision. Han et al. [82] adopted a greedy approach that continuously adds/removes one server to the most/least utilized tier until the estimated capacity is just enough to serve the current load.

As mentioned in Section 2.4.9, some parameters in the queuing models are hard to measure directly, like service time. Therefore, the proposed auto-scalers should properly handle this issue as well. The detailed techniques have already been introduced in Section 2.4.9.

Machine Learning

Machine learning techniques in resource estimation are applied to dynamically construct the model of resource consumption under a specific amount of workload (online learning). In this way, different applications can utilize the auto-scalers without customized settings and preparations. They are also more robust to changes during production as the learning algorithm can self-adaptively adjust the model on the fly regarding any notable events. The online machine learning algorithms are often implemented as feedback controllers to realize self-adaptive evolution. Though offline learning can also be used to fulfill the task, it inevitably involves human intervention and thus loses the benefit of using machine learning. For works that are using offline learning — if there exists any, we would prefer to classify them into the application profiling category.

Despite their easiness of usage and flexibility, machine learning approaches do suffer a major drawback. It takes time for them to converge to a stable model and thus causes the auto-scaler to perform poorly during the active learning period. Certainly, the application performance is affected in this process. Furthermore, the time that is taken to converge is hard to predict and varies case by case and algorithm by algorithm.

Online learning used by existing auto-scalers can be divided into two types: reinforcement learning and regression.

Reinforcement Learning: Reinforcement learning aims to let the software system learn how to react adaptively in a particular environment to maximize its gain or reward. It is suitable to tackle automatic control problems like auto-scaling [21,27,56,57,61,90,127,191,216,228]. For the auto-scaling problem, the learning algorithm's target is to generate a table specifying the best provision or deprovision action under each state. The learning process is similar to a trial-and-error approach. The learning algorithm chooses an indi-

vidual operation and then observes the result. If the result is positive, the auto-scaler will be more likely to take the same action next time when it faces a similar situation.

The most used reinforcement learning algorithm in the auto-scaling literature is Q-learning. A detailed description of the Q-learning algorithm and their variations in auto-scaling can be found in Section 5.2 of the survey by Lorido-Botran et al. [133].

Regression: Regression estimates the relationship among variables. It produces a function based on the observed data and then uses it to make predictions. Under the context of resource estimation, the auto-scaler can record system utilization, application performance, and the workload for regression. As the training proceeds and more data are available, the predicted results also become more accurate. Although regression requires the user to determine the function type first, for example, whether the relationship is linear or quadratic, in the case of auto-scaling web applications, it is usually safe to assume a linear function.

Chen et al. [37] used regression to dynamically build the CPU utilization model of Live Messenger given some active connections and the login rate. The model is then used for resource provision. Bodik et al. [23] employed smoothing splines nonlinear regression to predict mean performance under a certain amount of resources. Then they calculated the variance based on the estimated mean. After that they used a local polynomial (LOESS) regression to map mean performance to variance. Through this method, they found out that higher workload results in both mean and variance of the response time to increase. To detect sudden changes, they rely on conducting a statistical hypothesis test of the residual distribution in two different time frames with probably different sizes. Suppose the test result is statistically significant, the model needs to be retrained. Padala et al. [152] utilized auto-regressive-moving-average (ARMA) to dynamically learn the relationship between resource allocation and application performance considering all resource types in all tiers. Gambi et al. [67] proposed an auto-scaler using a Kriging model. Kriging models are spatial data interpolators akin to radial basis functions. These models extend traditional regression with stochastic Gaussian processes. The major advantage of them is that they can converge quickly using fewer data samples. Grimaldi et al. [77]

proposed a Proportional-Integral-Derivative (PID) controller that automatically tunes parameters to minimize integral squared error (ISE) based on a sequential quadratic programming model.

Yanggratoke et al. [215] proposed a hybrid approach using both offline learning and online learning. They first used a random forest model and traces from a testbed to train the baseline. Then they applied regression-based online learning to train the model for real-time resource estimation.

Hybrid Approaches

All the previous listed approaches have their advantages and limitations. Therefore, some works have integrated multiple methods together to perform resource estimation. We classify them as hybrid approaches and individually introduce them and the rationales behind such integration.

Rule-based approaches are inflexible when significant changes occur to applications and often require expert knowledge to design and test. However, if the rules can be constructed dynamically and adaptively by some learning techniques, such concern vanishes. Jing et al. [103] and Jamshidi et al. [95] proposed approaches that combine machine learning and fuzzy rule-based inference. They utilized machine learning to dynamically construct and adjust the rules in their fuzzy inference engine. Lama and Zhou [123] first proposed a fixed fuzzy-based auto-scaler with a self-adaptive component that dynamically tunes the output scaling factor. After that, they proposed another fuzzy inference approach as a four-layer neural network [122] in which the membership functions and rules can self-evolve as the time passes.

Some analytical queuing models require the observation of volatile metrics that are hard to measure directly. In these cases, a widely-adopted solution is to use machine learning approaches to estimate the concealed metrics dynamically. Gandhi et al. [69] adopted Kalman filter to assess the average service time, background utilization, and end-to-end network latency. Zhang et al. [223] employed application profiling and regression to learn the relationship of average CPU utilization and average service time at each tier under given request mix to solve their queuing network model using Mean

Value Analysis.

To mitigate the drawback of machine learning approaches, which are slow to converge and may cause plenty of SLA violations, another model can be used to substitute the learning model temporarily and then shift it back after the learning process has converged. Tesauro et al. [192] and Gambi et al. [66] proposed this type of auto-scalers. Both of them utilized an analytical queuing model for temporary resource estimation during the training period. Tesauro et al. [192] employed reinforcement learning while Gambi et al. [66] adopted a Kriging-based controller for training.

2.4.12 Oscillation Mitigation

Oscillation is the situation that auto-scaler continuously performs inverse scaling operations back and forth, such as provisioning 2 VMs and then in short time deprovisioning 2 VMs. It happens when monitoring and scaling operations are too frequent, or the auto-scaler is poorly configured. Such concerns are magnified when dealing with rule-based auto-scalers whose resource estimations are relatively empirical and coarse-grained. If the scaling thresholds are poorly configured, oscillation is likely to happen. For example, suppose the scale-up threshold is set to 70%, the scale-down threshold is set to 50%, and the current utilization is 71% with only one instance running, the auto-scaler will add one more instance to the cluster to reduce the utilization. It then quickly drops to 35%, which is below the scale-down threshold, thus causing oscillation.

Cooling Time

One common solution adopted by industries [11] to mitigate oscillation is to coercively wait a fixed minimum amount of time between each scaling operations. The time is set by users and is widely called as the cooling time. It should be set to at least the time taken to acquire, boot up, and configure the VM. Such method is simple but effective to avoid frequent scaling operations. However, setting a long cooling time will also result in more SLA violations as the application cannot be scaled up as quickly as before. Besides, it cannot handle the situation that the auto-scaler is poorly configured.

Another way of setting the cooling time is to confine the scaling condition further. Suppose the monitoring interval of the auto-scaler is 1 minute, we can achieve a prolonged scaling interval by setting the scaling trigger to how many times the monitored value exceeds the defined threshold consecutively.

Dynamic Parameters

Besides cooling time, researchers have proposed approaches that dynamically adjust some parameters to reduce the possibility of causing oscillation.

Lim et al. [128,129] described an approach through dynamically tuning the triggering thresholds for scale-down operations. The core idea is to increase the scale-down threshold when more resources are allocated to decrease the target utilization range and vice versa when resources are deallocated, which can effectively mitigate oscillation if the application resource requirement varies significantly during peak time and non-peak time. Usually, during the non-peak time, a large target range is desirable to avoid the situation described in the poorly configured example, while during peak hours, a small target range is preferred to keep the utilization as close to the scale-up threshold as possible.

Bodik et al. [26] introduced a mechanism that they call "hysteresis parameters" to reduce oscillation. These parameters control how quickly the controller provisions and deprovisions resources. They are determined by simulations using Pegasus, an algorithm that compares different control settings to search the suitable one. Pralada at al. [152] used a stability factor to adjust the aggressiveness of the auto-scaler. As the factor increases, the control objective will be more affected by the previous allocation. As a result, the auto-scaler responds more slowly to the produced errors caused by the previous actions in the following resource scaling windows and thus reduces oscillations. Lama and Zhou [123] employed a similar approach on their fuzzy-based auto-scaler. Their approach is more advanced and flexible as the factor is self-tunable during runtime according to the resulted errors.

2.4.13 Theory

The above methods are only capable of mitigating the possibility of oscillations. If in theory, we can identify the settings that might cause oscillations and thus pose restrictions on such settings, the risk of oscillation will be eliminated. Cunha et al. [49] and Netto et al. [145] adopted this approach and proposed models that identify the potential oscillation conditions in their rule-based auto-scalers.

2.4.14 Scaling Timing

When to scale the application is a critical question needed to be answered by auto-scalers. However, there is no perfect solution for this issue as different applications have diverse workload characteristics, and preference of cost and QoS. Auto-scalers can be classified into two groups based on this criterion: auto-scalers that reactively scale the application only when necessary according to the current status of the application and the workload, and auto-scalers that support proactively provision or deprovision resources considering the future needs of the application.

For applications with gradual and smooth workload changes, reactive auto-scalers are usually preferred because they can save more resources without causing a significant amount of SLA violations. In contrast, applications with drastic workload changes or strict SLA requirements often require proactive scaling before the workload increases to avoid incurring a significant amount of SLA violations during the provisioning time. Such strategy relies on prediction techniques to timely foresee incoming workload changes. Prediction is the process of learning relevant knowledge from the history and then apply it to predict the future behaviors of some object. The assumption that behaviors are predictable lies that they are not completely random and follow some rules. Therefore, workload prediction is only viable for the workloads with patterns and thus, cannot handle the random bursts of requests, which is common in some applications, like news feed and social network. For these bursty workload scenarios, currently there is no effective solution, and we can only deal with them reactively in the best effort. Hence, regardless the existence of support for proactive scaling, a qualified auto-scaler should always be

able to scale reactively.

Proactive Scaling

As the accuracy of the prediction algorithm determines the capability of the auto-scaler to scale applications proactively, in this section, we survey prediction algorithms that have been employed by state-of-the-art works.

Workload Prediction Data Source: It is necessary to study the past workload history to understand workload characteristics, including the workload intensity and workload mix during each time frame, to predict the workload. General purpose workload predictors usually only utilize past workload information to make predictions.

Besides workload history, individual applications can rely on available information from other aspects to predict request bursts that are impossible to be derived from past workload data alone, such as weather information for an outdoor application, and political events for a news feed application. However, the relevant parameters are application specific and thus this feature is hard to be integrated into a general purpose auto-scaler. Besides, it is also challenging to devise a prediction algorithm with real-time accuracy for resource provisioning, because there are too many parameters in the model and errors can quickly accumulate. The work by Frey et al. [65] considers multiple outside parameters in an auto-scaler. Their approach integrates all the prediction information into a fuzzy controller.

Though it is challenging to predict the right amount of workload with outside information, it is viable to timely detect events that may affect incoming workload intensity through social media and other channels [219]. Since this is a broad topic itself, we focus on prediction algorithms only based on workload history.

Prediction Horizon and Control: Typically, a prediction algorithm loops in a specified interval to predict the average or maximum workloads arriving at the application during each of the next few intervals, which form the prediction horizon. It determines how far in the future the auto-scaler aims to predict.

There are two approaches that auto-scalers can apply the prediction results in resource provision. The first way, which is adopted by the majority of works, takes the prediction horizon as the control interval and scales the application only based on the predicted workload of the next horizon. The weakness of this approach is that the auto-scaler is likely to make short-sighted scaling decisions if the horizon is too short or the volatility of workload is high. The other strategy is called Model Predictive Control (MPC). It sets the control interval the same to the prediction interval. When making decisions, it considers all the intervals within the horizon and determines the scaling operations at each interval using optimization. However, when executing the scaling operations, it only performs the action for the next interval and discards operations for the rest intervals in the horizon. This method mitigates the problem of provision for short-term benefits, but it requires solving complex optimization models, and thus, consumes much more computing power. Ghanbari et al. [73, 74], and Zhang et al. [224] employed this approach.

To tune the length of the horizon, users can either adjust the duration of each interval or number of intervals in the horizon. The size of the interval is critical to prediction precision. A large interval can significantly degrade the prediction accuracy and is useless for real-time control if the interval is greater than the control interval of the auto-scaler. The number of intervals in the horizon is also a crucial parameter, especially for the MPC approach. A balanced number should be chosen for the auto-scaler to reach good performance. If it is too small, MPC cannot fully realize its potential to make decisions for the long-term benefit. A large number, on the other hand, may mislead the auto-scaler as predictions for the intervals far in the future, become increasingly inaccurate.

Workload Prediction Algorithms: Regarding workload prediction algorithms, they can be coarsely classified into two types: prediction according to recent trends and prediction based on regular patterns.

Prediction according to recent trends aims to use the workload data monitored in the near past to determine whether the workload is increasing or decreasing and how much it will change. In this case, only a few data is stored for prediction purpose. Time

series analysis algorithms are commonly applied to this type of prediction tasks, such as linear regression [26], various autoregressive models (AR) [37,62,164,214,216], and neural network-based approaches [16,147,156]. Besides using time-series analysis, Nguyen et al. [146] proposed another method, which considers each time interval as a wavelet-based signal and then applies signal prediction techniques.

Prediction algorithms based on regular patterns assume the workload is periodic, which is valid for many applications as they tend to be more accessed during the daytime, weekdays, or the particular days in a year (tax report period, Christmas holidays). By finding these patterns, predictions can be easily made. Different from prediction algorithms based on recent trends, this type of algorithm requires a large workload archive across an extended period. Various approaches have been explored to identify workload patterns when building auto-scalers. Fang et al. [62] employed signal processing techniques to discover the lowest dominating frequency — the longest repeating pattern. Silva Dias et al. [50] utilized Holt-Winter model, which aims to identify seasonality in the workload for prediction. Jiang et al. [100] devised an approach by first identifying the top K most relevant monitored data using an auto-correlation function and then employing linear regression on the selected data for prediction. Urgaonkar et al. [202] adopted an algorithm based on the histogram for the workload with daily patterns.

Herbst et al. [87] integrated many predictors into one auto-scaler. They presented an approach to dynamically select appropriate prediction methods according to the extracted workload intensity behavior (WIB, simply the workload characteristics) and user's objectives. The mappings of prediction methods to WIBs are stored in a decision tree and are updated during runtime based on the recent accuracy of each algorithm.

Resource Usage Prediction: Instead of predicting workload, it is also possible to directly predict resulted resource usage according to the historical usage data. This strategy is commonly used by auto-scalers only support vertical scaling, as for a single machine, resource usage can substitute workload intensity. Some proposals [32,93] that target horizontal scaling also follows this strategy to accomplish both workload prediction and resource estimation together.

Gong et al. [76] used signal processing to discover the longest repeating pattern of resource usage and then relied on dynamic time warping (DTW) algorithm to make the prediction. For applications without repeating patterns, they referred to a discrete-time Markov chain with finite states to derive a near prediction of future values. Islam et al. [93] explored the use of linear regression and neural network to predict CPU usage. Caron et al. [32] adopted a pattern matching approach which abstracts it as a string matching problem and solved it using the Knuth-Morris-Pratt (KMP) algorithm. Yazdanov et al. [217] utilized an auto-regressive (AR) method to predict short-term CPU usage. Almeida Morais et al. [6] employed multiple time series algorithms to predict CPU usage, and based on their runtime accuracy, the best prediction algorithm is selected. [132] also used various prediction algorithms. However, instead of selecting the best one, their approach combine the results of different predictors using weighted k-Nearest Neighbors algorithm. The weight of each predictor is dynamically adjusted according to their recent accuracy.

2.4.15 Scaling Methods

Depending on the particular Cloud environment, elastic scaling can be performed vertically, horizontally, or in a hybrid. Each of them has their advantages and limitations. In this section, we discuss the key factors that need to be considered when making the provisioning plan [88].

Vertical Scaling — VM Resizing

Vertical scaling means removing or adding resources, including CPU, memory, I/O, and network, to or from existing VMs. To dynamically perform these operations, modern hypervisors utilize mechanisms such as CPU sharing and memory ballooning, to support CPU and memory hot-plug. However, major Cloud providers, such as Amazon, Google, and Microsoft, do not support adjusting resources during runtime. In these platforms, it is essential to shut down the instance first to add resources. Some providers like Centu-

rylink[1] allow users to scale CPU cores without downtime vertically. Profitbricks[2] permits to add both CPU and memory to the VMs dynamically.

Vertical scaling is considered not suitable for highly scalable applications due to its limitations. Ideally, the maximum capacity a VM can scale to is the size of the physical host. However, multiple VMs are usually residing on the same physical machine competing for resources, which further confines the potential scaling capability. Though limited, dynamic vertical scaling outperforms horizontal scaling in provision time as it can be in effect instantaneously. Besides, some services or components that are difficult to replicate during runtime, such as database server, and stateful application server, can be benefited by vertical scaling. Dawoud et al. [52] conducted an experimental study of vertical scaling using RUBBOS benchmark on both its application server and database, which highlights the advantages of vertical scaling mentioned above.

Many auto-scalers have been developed using solely vertical scaling to manage VMs on the same physical host. Some of them only considered scaling CPU resources [109, 173,181,217], and others targeted both CPU and memory [52,76,216,228]. Jing et al. [103] focused on CPU and claimed their method could be extended to other resources. Bu et al. [27] proposed an approach that adjusts not only CPU and memory allocation but also application parameters. Padala et al. [152] scaled both CPU and disk. These auto-scalers are mostly deployed in private Clouds or by Cloud providers.

Horizontal Scaling — Launching New VMs

Horizontal scaling is the core of the elasticity feature of Cloud. Most Cloud providers offer standardized VMs of various sizes for customers to choose. Others allow users to customize their VMs with a specific amount of cores, memory, and network bandwidth. Besides, multiple pricing models are co-existing in the current Cloud market, which further increases the complexity of the provisioning problem.

[1] https://www.ctl.io/autoscale/
[2] https://www.profitbricks.com/help/Live_Vertical_Scaling

Heterogeneity: Regarding a single tier/service within a web application, if the billing is constant, the use of homogeneous VMs is well acceptable as it is easy to manage. The auto-scaling services offered by Cloud providers only allow the use of homogeneous VMs. Selecting which type of VM is considered the responsibility of users in commercial auto-scalers. The optimal solution depends on the resource profile of the tier/service, e.g., whether it is CPU or memory intensive, and the workload characteristic. If the workload is always large enough, different sizes of instances make little difference. While for a small and fluctuant workload, smaller instances are preferred as scaling can be conducted in finer granularity and thus save more cost.

Cost-efficiency of VM is highly co-related to the application and workload. If changes happen to them, the choice of VM type should also be reconfigured. Grozev and Buyya [79] proposed a method that detects changes online using the Hierarchical Temporal Memory (HTM) model and a dynamically trained artificial neural network (ANN) and then reselects the most cost-efficient VM type.

The use of heterogeneous VMs to scale web applications has been explored in the literature. Under conventional billing, where price grows linearly with VM's capability, heterogeneity can bring some extra cost-saving but not significant. Furthermore, it is often computing-intensive to search the provision plan with a combination of heterogeneous VMs. Srirama and Ostovar [182] employed linear programming to solve the provision problem, yet only achieved limited cost saving against AWS auto-scaling. Fernandez et al. [63] abstracted the provision combinations as a tree and searched the proper provision by traversing the tree according to different SLAs. In a different scenario in which the capability of VMs increases exponentially to their prices, heterogeneity has the potential to save significant cost, which is shown in works done by Sedaghat et al. [170] and Upendra et al. [200]. They employed a similar approach by considering the transition cost (the time and money spent to convert from the current provision to the target provision) and the cost of resource combination in the optimization problem.

Pricing Models: The current Cloud pricing models can be classified into three types by pricing model: on-demand, reserved, and rebated. In on-demand mode, the provider

sets a fixed unit price for each type of VM or unit of particular resource and charges the user by units of consumption. Users submit requests for resources and obtain the required resources with agreed performance. The resources are released only when the users terminate them, which most auto-scalers assume the target application is adopting. The reserved mode requires the user to pay an upfront fee for cheaper use of a certain amount of resources within an agreed period. If highly utilized, users can save a considerable sum of money than acquiring resources in on-demand mode. Providers create the rebated mode aiming to sell their spare capacity. They are usually significantly cheaper than on-demand resources. There are several ways to offer rebated resources. Amazon employed an auction mechanism to sell instances, called spot instances. In this mode, the user is required to submit a bid on the resources. Suppose the bid exceeds the current market price, the bid is fulfilled and the user is only charged for the current market price. The acquired spot instances are guaranteed to have the same performance of their on-demand counterparts. However, they are reclaimed whenever the market price goes beyond user's bidding price. Google offer their spare capacity as preemptible VMs. Different from Amazon, they set a fixed price to the VM, which is 30% of the regular price, and the VM is available at most for 24 hours. Rebated instances are considered not suitable to host web applications that are availability-critical. ClusterK[3] and our proposed approach in Chapter 5 however have demonstrated that it is feasible to build an auto-scaler utilizing spot instances by exploiting various market behaviors of different spot markets to achieve both high availability and considerable cost saving.

Pricing models also can be classified according to billing period, which is the minimum unit consumption. Providers have set their billing period to every minute, hour, day, week, month, or year. The length of the billing period has a significant impact on the cost-efficiency for elasticity. Obviously, the shorter the billing period, the more flexible and cost-efficient it is for auto-scaling.

[3]http://www.geekwire.com/2015/amazon-buys-clusterk-a-startup-that-lets-developers-run-aws-workloads-more-cheaply/ acquired by AWS in 2015

Hybrid

As mentioned, horizontal scaling is slow in the provision and vertical scaling is confined by the resources available in the host. It is natural to employ vertical scaling and horizontal scaling together to mitigate these issues. The idea is to utilize vertical scaling when possible to quickly adapt to changes and only conduct horizontal scaling when vertical scaling reaches its limit. Urgaonkar et al. [202], Huber et al. [89], Rui et al. [165], and Yang et al. [214] followed this strategy.

Mixing vertical scaling and horizontal scaling can also bring cost benefit. Dutta et al. [58], and Gandhi et al. [68] explored optimization techniques to search for the scaling plan that incurs the least cost with a hybrid of vertical and horizontal scaling.

Vertical scaling and horizontal scaling can be separately applied to different components of the application as well since some parts such as database servers are difficult to be horizontally scaled. Nisar et al. [148] demonstrated this approach in a case study.

2.5 Summary

In this chapter, we reviewed the challenges and developments in both selection and provision aspects of managing web applications in multiple Clouds. Particularly, we focused on discovery and selection of Cloud services according to QoS requirements and auto-scaling techniques. Our proposal aims to enhance the state-of-the-art by addressing some tackled challenges or improving the previous solutions.

Regarding Cloud service discovery algorithms, we classify them into two broad groups: service ranking and matchmaking. We compared methods used in earlier works and identified their shortcomings. In the proposed solution (Chapter 3), we integrated hierarchical fuzzy inference into multi-criteria Cloud discovery to ease the representation of user requirements and preferences. It also considers performance variations by utilizing various statistical metrics.

For selecting data centers to host web applications, we listed major factors that are considered in the process. After that, we introduced existing works based on their specific goals and models. Different from the previous works, we take extra latencies caused

by complying data consistency semantics and data center migration cost into account in the proposed solution (Chapter 4).

About auto-scaling techniques, we abstracted them as a MAPE (Monitoring, Analysis, Planning, and Execution) loop. We identified challenges in each phase of the loop and presented a taxonomy of the existing works based on their solutions to the challenges. Upon that, we explained and compared the existing approaches. To further improve their cost-efficiency, we propose to utilize unreliable rebated resources to auto-scale web applications. Our approach is not only capable of saving significant cost, but also ensures high availability under current Cloud market (Chapter 5). In addition to auto-scaling, we further propose a solution that handles short-term resource overloads which are complicated to be dealt timely and effectively by current auto-scaling approaches, through geographical load balancing among multiple data centers (Chapter 6).

In the next chapter, we introduce the proposed technique for discovering satisfactory Cloud services.

Chapter 3

A Cloud Trust Evaluation Approach using Hierarchical Fuzzy Inference System for Service Selection

To realize the benefits of using Clouds, users need fist to select the proper Cloud services that can satisfy their applications' functional and non-functional requirements. However, this is a challenging task due to a large number of available services, users' unclear requirements, and performance variations in Cloud. Trust management systems can help users to identify adequate services in unstable environments. But existing trust evaluation approaches cannot be directly applied to the Cloud because of their limitations. In this chapter, we propose a new method that evaluates satisfiability of Clouds as trust according to users' fuzzy Quality of Service (QoS) requirements to facilitate service selection. We demonstrate the effectiveness and efficiency of our approach through simulations and show how our approach can be applied through case studies.

3.1 Introduction

TO realize the benefits of using Cloud, users need to ensure the trading service providers can fully satisfy their applications' functional and non-functional requirements. However, there are a plenty of Cloud providers offering similar services with different pricings and performances, which creates difficulty for users to find the most suitable service. Therefore, it is essential to develop automatic mechanisms to identify satisfactory services according to different application requirements.

This chapter is derived from: **Chenhao Qu**, and Rajkumar Buyya, "A Cloud Trust Evaluation System using Hierarchical Fuzzy Inference System for Service Selection", *Proceedings of the 28th IEEE International Conference on Advanced Information Networking and Applications (AINA 2014, IEEE CS Press, USA)*, Victoria, Canada, May 13-16, 2014.

Observed by experiments [169], VM performances within the same Cloud are far from stable. Schad et al. [169] pointed out that one influencing cause is the hardware heterogeneity in the underlying physical infrastructure, e.g., different types of CPU, memory, and disk used on the physical hosts. Besides that, the interference from collocated VMs also affects performance significantly [119]. Thus, the surge of VM requests in peak hours often causes performance degradation. Some other providers, e.g., eApps [59], dynamically allocate CPU to VMs on the same host according to their priorities, which makes VM performances even more unstable. Since the aggregated inherent variability is non-negligible [169], it is necessary to take it into account in the selection phase to improve the cost-efficiency of using Clouds.

Trust management systems have successfully helped users to select competent and trustworthy services in different dynamic environments [106]. However, existing trust evaluation approaches cannot be directly applied to the Cloud. The major obstacle is that Cloud users have different expectations on service when deploying various applications. Therefore, Cloud trust evaluation systems should be able to capture personalized requirements and preferences to provide customized service.

Many state-of-art Cloud service discovery approaches [71, 142, 197–199, 209] need users to submit static weights to model preferences for attributes, which requires expert knowledge and is time-consuming. To let users smoothly adopt Cloud, a more user-friendly way is to let them represent their vague preferences in linguistic phrases. Besides, sometimes it is also difficult for users to define QoS requirements in actual values, e.g., users need to conduct sophisticated tests to determine the exact CPU power required to process 50 transactions in parallel. Similarly, by using approximate linguistic descriptors, users can define requirements easily and quickly.

In this chapter, we propose a new method that evaluates and ranks the satisfiability of Cloud services to user's personalized requirements to support Cloud service selection. In particular, we measure trust of Clouds as their satisfactory degree to specific user requirements based on their past performances. We employ membership functions and fuzzy hedges to capture users' personal requirements and preferences for different QoS attributes and then use a hierarchical fuzzy inference system to derive trust levels. By

analyzing past benchmark results, our approach can identify services that are likely to meet all QoS requirements in the whole application life cycle. Not that, like other Cloud service discovery approaches, our approach is general-purpose and can be applied to various kinds of applications apart from web applications. Through simulations experiments, we demonstrate the effectiveness of our approach, and by case studies, we show how our approach can be applied.

The remainder of this chapter is organized as follows. We introduce relevant background about trust management and fuzzy inference in the next section. Then we discuss related works and their limitations. After that, we present our trust evaluation method, followed by the performance evaluation. In Section 3.6, we illustrate the usage of our approach with two case studies. Then we discuss the limitations of our approach in Section 3.7. Finally, we summarize the chapter.

3.2 Background

3.2.1 Trust Management System

Trust has different definitions under particular contexts. In distributed system, trust is usually defined as the subjective belief that the system the user intending to interact with will behave as expected [106]. Such belief should be derived from substantial evidence, such as past performance, peer recommendations, and certificates. Trust management systems are designed to aggregate trust from above evidence in real time and provide trust query services to parties in concern. They have played important roles in helping users to identify adequate services in different environments. However, since state-of-the-art trust management systems are usually developed for specific platforms, they are unsuitable for Cloud where users run diverse applications for various purposes. Besides, they often use user ratings as evidence, which is also inappropriate for Clouds because users' different expectations for services are likely to affect their ratings. For existing trust management systems and trust evaluation techniques, interested readers can find more details in the survey done by Jøsang et al. [106]. Different from previous methods in other environments, our trust evaluation approach can provide customized service according

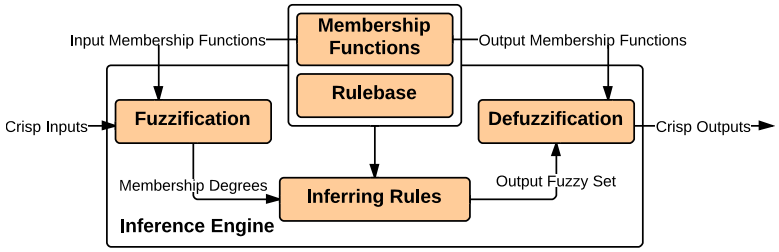


Figure 3.1: A Typical Fuzzy Inference System

to users' individual expectations.

3.2.2 Hierarchical Fuzzy Inference System

Fuzzy inference systems have been widely used to solve control and reasoning problems in uncertain environments due to its ability to handle inaccurate inputs. Figure 3.1 shows a typical fuzzy inference system. It has three main elements:

1) **Inference Engine:** It defines the fuzzy logic operators and defuzzifier used in the inference process.

2) **Membership Functions:** A membership function determines to what degree the fuzzy element belongs to the corresponding fuzzy set. It maps crisp values to membership levels between 0 and 1. In fuzzy inference system, each input and output variable have its individual set of membership functions.

3) **Rulebase:** It is a set of "If-Then" rules that define the inference model. The rule structure is like: "If *antecedent* Then *consequent*", where *antecedent* and *consequent* are fuzzy propositions connected by "AND" or "OR" operators.

The inference process usually involves five major steps:

1) **Fuzzification:** input crisp values into the membership functions to obtain corresponding membership degrees of each input variable regarding specific fuzzy set.

2) **Applying Fuzzy Operations:** obtain the membership degree of the *antecedent* using "AND" and "OR" operators.

3) **Implication:** obtain the fuzzy set of each rule using the defined implication operator.

4) **Aggregation:** aggregate output fuzzy sets of all rules using the defined aggregation operator.

5) **Defuzzification:** transform the aggregated fuzzy set into a crisp value using the defined defuzzification algorithm.

A hierarchical fuzzy inference system is connected by multiple atomic fuzzy inference modules with outputs of the low-level modules serving as inputs to the high-level modules. It brings two main advantages comparing to a non-hierarchical one. The first is that it can reduce the number of "If-Then" rules, which greatly simplifies its design. The second benefit is that it enables the system to compute partial solutions when the task can be clearly partitioned or there are functional dependencies in the system. For further information on these systems, readers can refer to Torra's survey [194].

3.3 Related Work

For service ranking approaches to discover Cloud services, many teams [71, 142, 197–199,209] have investigated using Multi-criteria Decision Making (MCDM) methodologies to rank Clouds. These approaches depend on static weight assessment for preference modeling, which is time-consuming. Furthermore, except the work by Rehman et al. [199] and Wang et al. [209], none of them considered performance variations. Alabool et al. [2] developed a fuzzy based MCDM approach that uses linguistic descriptors to model preferences. Still, they did not address the performance variation problem. Noor et al. [150] and Habib et al. [80] evaluated trust of Clouds according to user feedbacks, but they ignored users' diverse requirements. Supriya et al. [186] also employed hierarchical fuzzy inference system to evaluate trust of providers. However, they required users to manually tune the inference system for each query regarding their expectations.

Dastjerdi et al [51] proposed the first influential Cloud service matchmaking approach. They adopted description logic to match user's QoS goal and services' self-advertised Service Level Agreement (SLA) contracts. Sundareswaren et al. [185] proposed a time-

efficient selection algorithm for Cloud brokers based on B+ tree indexing. Redl et al. [160] employed Support Vector Machine algorithms to find the closest Cloud service to user requirements. None of them considered performance variations, and only the approach by Sundareswaren et al. [185] can model user preferences.

Different from previous works, our approach uses fuzzy linguistic descriptors and hedges to help users to quickly and easily define their requirements and preferences, and it considers performance variations in Clouds when evaluating trust, which improves cost-efficiency for Cloud users.

Apart from our work, fuzzy logic has been applied to address other service discovery problems. Nepal et al. [144] employed fuzzy set operations and fuzzy hedges for preference modeling in their web service discovery approach. Wang [208] used a fuzzy based MCDM algorithm to rank web services. Song et al. [179][180] utilized fuzzy inference system to evaluate trust in Grid and P2P systems for resource selection.

3.4 Proposed Approach

3.4.1 Architecture

Figure 3.2 illustrates the general architecture of our proposed approach. There are two major steps involved in it. The first step, which is shown in dashed lines, captures users' subjective perceptions of different QoS attributes through tuning fuzzy membership functions. The second step, which is shown in solid lines, involves the whole process to evaluate Cloud services.

The components of the architecture are explained below:

Web Interface

This layer provides users or Cloud brokers with the entrance to the service. Users can submit their functional and non-functional requirements, and change their perceptions through graphical interfaces.

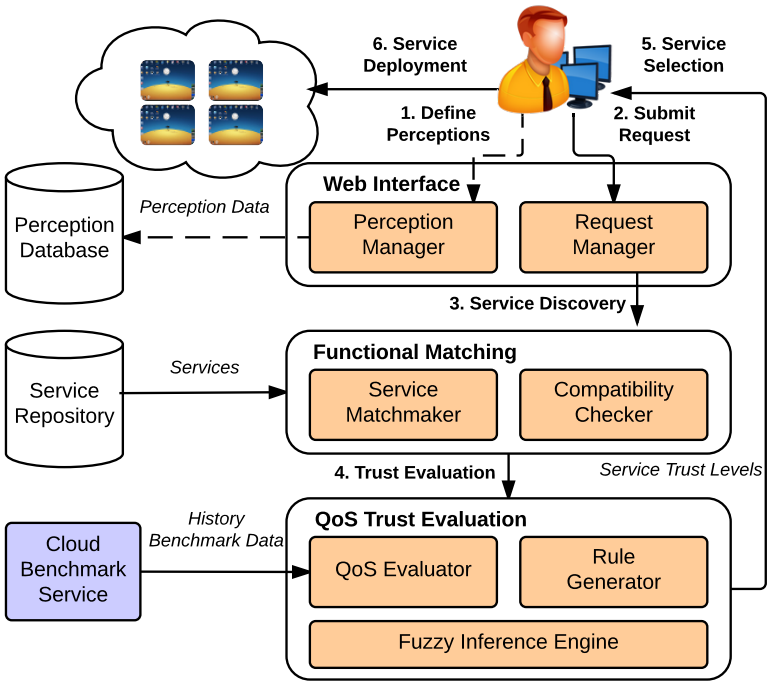


Figure 3.2: Architecture of the Proposed Trust Evaluation Approach

Functional Matching

This component retrieves services that can meet users' functional requirements (i.e., the number of cores, memory amount, storage volume, budget and geographical location) and some static QoS requirements (e.g., security and privacy) from the service repository. Another important function of it is to check the compatibility of services. It filters services that cannot satisfy users' business policies or are incompatible to software platforms required by their applications. Several tools and techniques have been developed for these purposes, such as the approaches proposed by Dastjerdi et al [51] and Chen et al. [36].

QoS Trust Evaluation

It is the core of our approach that evaluates the trust levels of functionally matched services. It takes user requirements and the services' past benchmark results as input and then outputs a list of services with their trust values regarding each attribute. Users or Cloud brokers can then select the most suitable services based on the obtained trust val-

ues along with other objectives (e.g., cost and location).

Cloud Benchmark Service

These services continuously monitor the performances of Clouds by running benchmark applications on some dynamically launched VMs in a particular time interval and publishing the results to the public. They provide strong data traces regarding low-level metrics that are required to compare Cloud services' QoS reasonably. An example of such service is Cloud Harmony [84]. By the time of this work, it was monitoring 63 Cloud data centers all over the world.

3.4.2 Modelling Requirements and Preferences

Requirement Types

Our approach requires users to submit requirements for attributes that are relevant to them. It supports two types of requirements, namely, numerical requirement and linguistic requirement. Submission of mixed types of requirements in the same query is valid in our approach. Section 3.6 illustrates how different types of requirements can help users to define their expectations in the discovery process with two case studies.

Numerical Requirements: Numerical requirements are numerical values with corresponding units. When submitting these requirements, users expect the services to have higher performances than the provided values. The numerical requirements for some attributes, e.g., availability, can be easily extracted from the application's Service Level Objectives (SLOs). While for other attributes, e.g., CPU and network speed, they cannot be effortlessly determined because performance SLOs are usually defined in high-level metrics, e.g., response time and throughput. This requires users to perform tests to transform high-level SLOs to low-level requirements.

Linguistic Requirements: Linguistic requirements are submitted as fuzzy linguistic descriptors (e.g., *High*, *Medium*, and *Low*) which are semantic approximations of the nu-

merical requirements. Since human beings are accustomed to using these linguistic descriptors to make rough estimations, they can quickly submit approximate requirements in this form according to application nature or preliminary results obtained through simple tests or simulations.

QoS Attributes

Table 3.1 shows an example of a hierarchy of Cloud QoS attributes, which we use in the prototype. We choose the attributes and their metrics according to the SMI framework [48] and the work by Garg et al. [71]. One can easily add extra attributes to the example model or use different metrics to measure existing attributes, e.g., IOPS (Input/Output Operations Per Second) for memory and disk performances. Furthermore, our approach allows users to selectively submit requirements for the attributes they concern. In such cases, it implicitly recognizes performances of the overlooked attributes to be entirely satisfactory for all services in the evaluation.

In general, we classify all leaf attributes in the hierarchy into two categories, namely, dynamic attributes and static attributes. Dynamic attributes suffer from performance variations. Therefore, their performances need to be quantified by benchmark traces. Compared with these attributes, performances of other attributes can be considered static, i.e., security attributes, as their variations are negligible and immeasurable. Our approach quantifies the performances of security attributes in the form of single values instead of series of traces. They can be evaluated by Cloud security benchmarks, e.g., the framework proposed by Garcia et al. [135], or expert ratings. We only allow users to submit linguistic requirements for static attributes, as for them, services can be binarily filtered with given numerical thresholds.

Besides QoS, the cost is also an important factor. Our approach provides three different ways to balance users' QoS and cost objectives. The first way is to specify a budget at functional matching phase. In this way, users can identify the most satisfactory service within an acceptable budget. For the second method, users can select the most economical Cloud service among those that have acceptable trust levels. The third way is to submit linguistic requirements for cost during the trust evaluation phase if users only

Table 3.1: An Example of Hierarchy of Cloud Attributes and Metrics for Trust Evaluation

Attributes			Metrics	Types	Requirements
Performance	CPU	CPU Speed	Maximum number of instructions executed by a single core in unit time (MIPS)	Dynamic	Numerical
	Memory	Memory Read	Maximum amount of data transferred from memory in unit time (Mb/s)		
		Memory Write	Maximum amount of data written to memory in unit time (Mb/s)		
	Storage	Disk Read	Maximum amount of data transferred from secondary storage in unit time (Mb/s)		
		Disk Write	Maximum amount of data written to secondary storage in unit time (Mb/s)		
	Network	Inbound	Maximum amount of data transferred into VM in unit time (Gbit/s)		
		Outbound	Maximum amount of data transferred out of VM in unit time (Gbit/s)		Linguistic
Assurance	Availability		The proportion of time that the VM is accessible (%)		
	Failure Rate		Average number of failures for one VM in one hour		
Elasticity	VM Startup Time		Time consumed to allocate, boot up, and configure VM (s)		
	VM Shutdown Time		Time consumed to shut down and deallocate VM (s)		
Security	Platform Security		Score of security mechanisms that protect virtualized platform	Static	Linguistic
	Data Security		Score of security mechanisms that protect users' data		
	Network Security		Score of security mechanisms that protect VMs from network attacks		
	Site Security		Score of security mechanisms that protect the data center		
	Security Policy		Score of policies that users can employ to implement their own security strategies		
Cost	VM Cost		The cost to deploy a VM ($/hr or $/month)		
	Data Transfer Cost		The cost to transfer data in or out of data center ($/Gb)		
	Storage Cost		The cost of secondary storage ($/Gb/Month)		

have vague objectives for the cost.

QoS Input and Membership Functions

As the first step of trust evaluation, our approach retrieves corresponding benchmark results from Cloud benchmark services and then analyzes the traces to derive the QoS input of the hierarchical fuzzy inference system according to user requirements. Regarding the two types of requirements, we respectively introduce how our approach calculates QoS inputs from benchmark traces and their associated input membership functions.

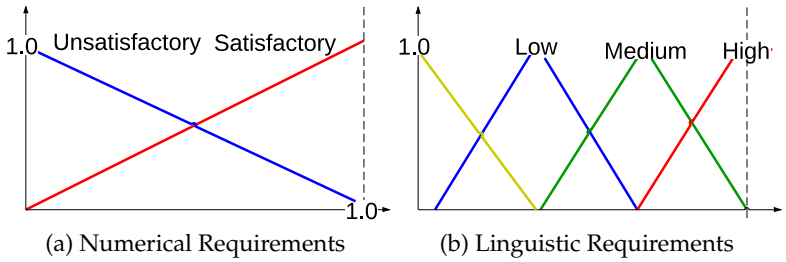
Retrieving Benchmark Traces: The benchmark traces used are key to the quality of trust evaluation. To retrieve representative traces, the evaluation process should consider multiple factors. Our approach selects traces according to the variability of attributes and expected running time of VMs.

Different attributes have various levels of variability. According to Schad et al. [169], VM startup time has very high variability in short time. For such attributes, it should retrieve traces within a short time window, e.g., 10 minutes.

The expected running time is also important when retrieving traces. Suppose a user wants to deploy a VM on Wednesday from 7:00 am to 2:00 pm, we should refer to traces that were benchmarked on nearest Wednesdays or weekdays during the same range of time. This is because Clouds are likely to suffer more variations at working hours on weekdays in their local time. Furthermore, if the user plans to deploy VMs for a long term, it is also better to consider the traces within a relatively larger time window to ensure the selected service is likely to be satisfactory in the whole life cycle of the application.

Numerical Requirements: For numerical requirements, our approach calculates the QoS inputs of the i_{th} service regarding the j_{th} attribute as follow:

$$p_{i,j} = \frac{{n^{satisfy}}_{i,j}}{{n^{total}}_{i,j}} \tag{3.1}$$



(a) Numerical Requirements (b) Linguistic Requirements

Figure 3.3: Membership Functions for Hierarchical Fuzzy Inference System

where for the performance of i_{th} service regarding the j_{th} attribute, the numerator and denominator of the equation respectively stand for the number of benchmark traces that can satisfy the numerical requirement and the total number of traces retrieved. In fuzzy inference process, the membership functions associated with QoS inputs of numerical requirements are shown in Figure 3.3a.

Linguistic Requirements: For linguistic requirements, our approach calculates statistical indicators of the benchmark traces as the QoS inputs. In Section 3.5, we test it with different indicators, i.e., median, mean, first quartile, and five percentile.

We use triangular membership functions to model linguistic requirements in fuzzy inference. Figure 3.3b shows an example of the membership functions. They represent users' personal perceptions that how quantitative performance data are mapped to qualitative linguistic descriptors for each attribute. Since perceptions are subjective, we allow users to individually adjust the functions based on the default ones defined by experts.

Modelling Preferences

In our approach, users can express their preferences by selecting one of the importance levels in Table 3.2 for each requirement, which allows users to make trade-offs among their requirements. For example, suppose a user wants to deploy a security sensitive application on Cloud, he might prefer to select the services that can satisfy his security requirements even at the cost of enduring more performance degradation for other attributes within an acceptable level. In this case, he can submit *Very Important* for security

Table 3.2: Importance Levels and Importance Coefficients

Importance Level	Importance Coefficient
Very Important	0.5
Important	0.66
Neutral	1
Unimportant	1.5
Very Unimportant	2

requirement. To increase the room for the trade-off, he can provide negative importance levels, e.g., *Unimportant*, for requirements that he is willing to tolerate low performance. We implement this mechanism using linguistic hedges, which are adapter functions that change the shapes of original membership functions. Following equations show the formal definitions of importance levels for the two types of requirements:

$$Numerical : d = \begin{cases} satisfactory(x)^{a_i} \\ unsatisfactory(x)^{\frac{1}{a_i}} \end{cases} \tag{3.2}$$

$$Linguistic : d = \begin{cases} f(x)^{a_i} & \text{if } x \leq peak \\ f(x)^{\frac{1}{a_i}} & \text{if } x > peak \end{cases} \tag{3.3}$$

where $satisfactory(x)$ and $unsatisfactory(x)$ respectively represent $satisfactory$ and $unsatisfactory$ membership functions shown in Figure 3.3a. $f(x)$ is a triangular membership function shown in Figure 3.3b, and $peak$ is the x value of its maximum membership point. a_i is the importance coefficient of the i_{th} importance level. By applying positive importance hedges, e.g, *Important*, it further penalizes services that cannot fully satisfy the requirements in trust evaluation according to the amounts of performance deficiency. Oppositely, if negative hedges, e.g, *Unimportant*, are applied, it decreases the penalties. Table 3.2 shows the default importance levels and coefficients used in our prototype.

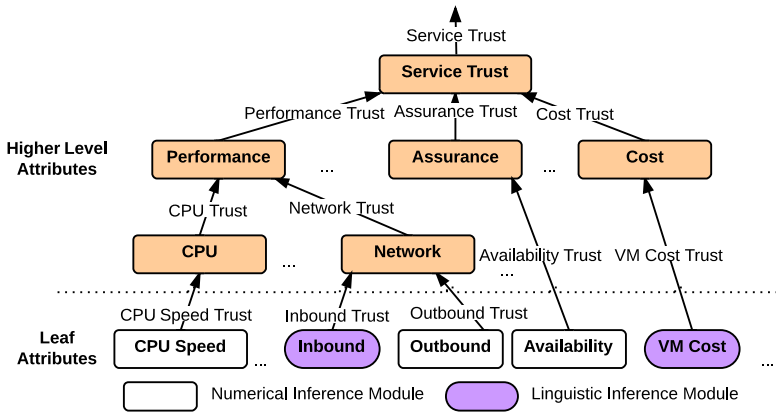


Figure 3.4: Example of Hierarchical Fuzzy Inference System for Trust Evaluation

3.4.3 Proposed Hierarchical Fuzzy Inference System

Overview

Our approach dynamically constructs the hierarchical fuzzy inference system for each query in agreement with the generated hierarchy of attributes. Figure 3.4 shows an example of it. There are three types of inference modules in the fuzzy system. For non-leaf attributes, their trust values are evaluated by higher level inference modules, which take outputs of corresponding sub-inference modules as inputs. At leaf level, it generates inference modules according to the types of requirements, namely, numerical inference modules and linguistic inference modules.

Higher level inference modules use functions shown in Figure 3.3a as input membership functions. The input membership functions for leaf level modules have been introduced in Subsection 3.4.2. All modules use the functions defined in Figure 3.3a as output membership functions. Rulebases for each module are generated according to user requirements, the details of which is introduced in the following part.

All inference modules share the same inference engine. Our prototype employs **Product** for "AND" operator, **MAX** for "OR" and Aggregation operator, and **Center of Maximum** (COM) defuzzifier. For each output variable with n membership functions, COM calculates the crisp trust value t as follow:

$$t = \frac{\sum_{i=1}^{n} x_i \mu_i}{\sum_{i=1}^{n} \mu_i} \tag{3.4}$$

where x_i is the x value of i_{th} membership function's maximum membership point and μ_i is the membership degree aggregated for the i_{th} membership function. The reason we choose COM is that it is both intrinsically plausible and timely efficient.

Generating "If-Then" Rules

Our approach dynamically generates the fuzzy rules for each module according to user requirements and employed rule generation strategy. In our prototype, we adopt a pessimistic strategy where output trust level is satisfactory only if all input variables are satisfactory, as we suppose users expect all submitted requirements should be individually satisfied. In the following subsections, we describe it in detail.

Higher Level Inference Modules: The inputs of these modules are trust levels of their corresponding sub-attributes. Suppose attribute A has sub-attributes B and C, the generated rule set is shown as follows:

Rule 1 If B_trust is $satisfactory$ AND C_trust is $satisfactory$ then A_trust is $satisfactory$

Rule 2 If B_trust is $not\_satisfactory$ OR C_trust is $not\_satisfactory$ then A_trust is $not\_satisfactory$

Leaf Level Inference Modules: For each leaf level inference module, the input is a single value obtained through the analysis of benchmark traces. If user submits numerical requirement for attribute A, the generated rules are as follows:

Rule 1 If A is ($importance\ level$) $satisfactory$ then A_trust is $statisfactory$

Rule 2 If A is ($importance\ level$) $not\_satisfactory$ then A_trust is $not\_satisfactory$

Otherwise, suppose user submits linguistic requirement *Medium* for attribute A, the resulted rule set is as follows:

Rule 1 If A is *at_least* (*importance level*) *Medium* then A_trust is *satisfactory*

Rule 2 If A is *at_most* (*importance level*) *Low* then A_trust is *not_satisfactory*

Low is the closest inferior linguistic descriptor to *Medium*. *at_least* is the linguistic hedge that transforms the membership function of *Medium* into function "$\geq$ *Medium*". Similarly, *at_most* transforms function of *Low* into "$\leq$ *Low*". The following equations show their formal definitions regarding triangular membership function $f(x)$:

$$d_{at\_least} = \begin{cases} f(x) & \text{if } x \leq peak \\ 1 & \text{if } x > peak \end{cases} \tag{3.5}$$

$$d_{at\_most} = \begin{cases} 1 & \text{if } x \leq peak \\ f(x) & \text{if } x > peak \end{cases} \tag{3.6}$$

where *peak* is the x value of the maximum membership point of the triangular membership function.

3.5 Performance Evaluation

3.5.1 Generating Benchmark Traces for Simulations

Schad et al. [169] ran performance benchmark applications (e.g., Ubench and Bonnie++) on newly launched VMs every hour in a month on Amazon EC2. They did not only quantify the performance variations regarding multiple attributes in a Cloud but also successfully identified their performance distributions. From the analysis of data, they found that the performance instability in Amazon is mainly caused by hardware heterogeneity and workload increase in peak time. Since these are common problems faced by all Cloud providers, we assume the performance distributions of other services also generally follow the same pattern as the one observed in Amazon but with different levels of variability and averages.

We test our approach through simulations using synthetic benchmark traces. Each service trace consists of 50 to 200 benchmark results for each of the 11 dynamic attributes

defined in Table 3.1. Some benchmark results are generated according to the known distribution patterns mentioned by Schad et al. [169]; others are generated using the normal distribution. In total, we created 3000 services with similar comprehensive performances but different performances regarding each attribute, which avoids our approach always returning the same set of omnipotent services and enables us to study how it performs with diverse user requirements.

3.5.2 Linguistic Requirements vs Numerical Requirements

In the first experiment, we test the validity and accuracy of using linguistic descriptors to approximate numerical requirements. To compare the results obtained by the two, we dynamically convert some numerical requirements in 25 numerical queries into linguistic requirements. This is conducted by selecting the linguistic descriptor that produces the highest membership degree with given numerical values and predefined membership functions. By doing so, we assume users can intuitively submit the closest linguistic descriptors to numerical requirements, which is the ideal situation.

We use average Normalized Discount Accumulative Gain (NDCG) at position 10 to measure whether our approach still can rank most satisfactory services at the top when some requirements are submitted in linguistic descriptors. The relevance scores used to calculate NDCG are trust levels derived from original numerical queries. Also, we measure absolute differences of trust levels obtained by the paired queries to gauge evaluation accuracy. Apart from the number of linguistic requirements, we also consider other factors that may influence accuracy, i.e., statistical indicators used to calculate the QoS inputs and number of linguistic descriptors (*Low*, *High* or *Low*, *Media*, *High*).

For each number of linguistic requirements, we run 55 tests with different combinations of attributes submitted in linguistic requirements. We use five linguistic descriptors for tests using different statistical indicators, and five percentile for tests using various numbers of linguistic descriptors. Figure 3.5 reports the average results of the tests.

According to the results, both the quality of ranking and evaluation accuracy decrease when the number of linguistic requirements grows, since they bring additional uncertainties. Also, the decreases are linear, which indicates limiting the number of lin-

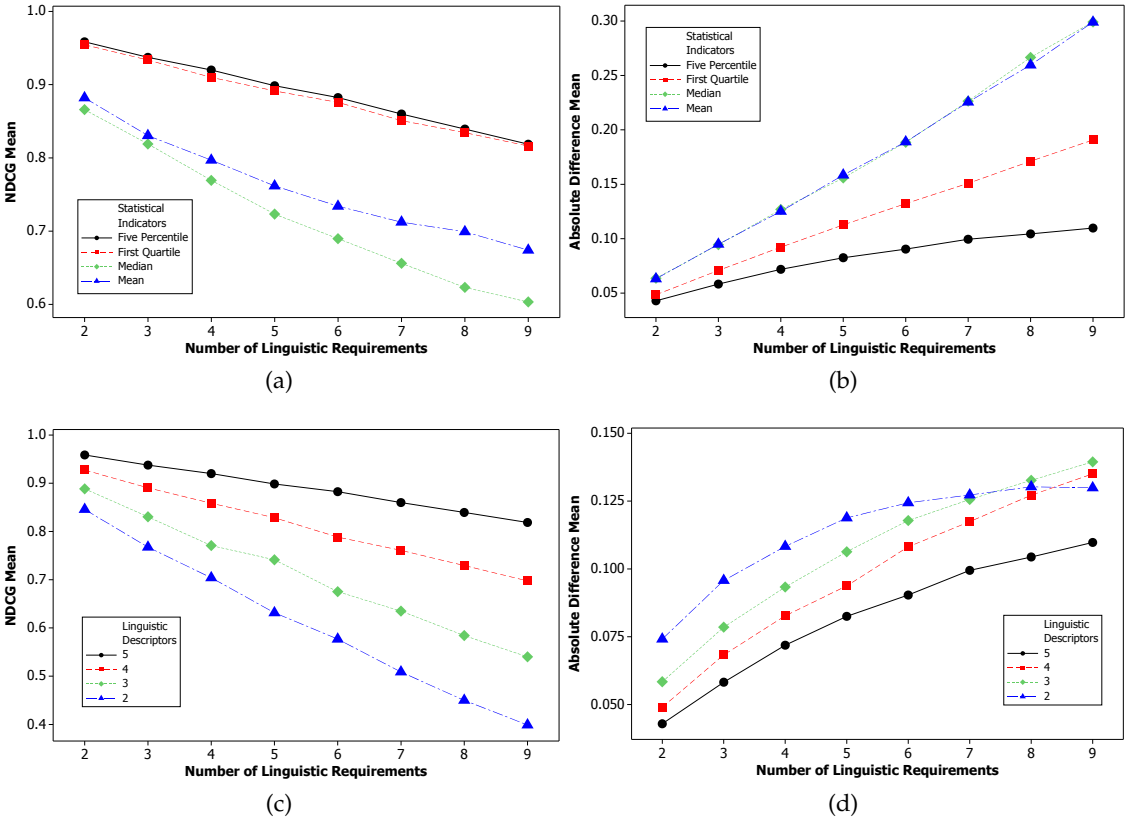


Figure 3.5: (a) NDCG mean and (b) absolute difference mean using different numbers of linguistic requirements and different statistical indicators. (c) NDCG mean and (d) absolute difference mean using different numbers of linguistic requirements and different numbers of linguistic descriptors.

guistic requirements can significantly improve the chance of finding the most satisfactory service. Comparing different statistical indicators, it shows that conservative indicators, i.e., five percentile and first quartile, produce better results, because they are more sensitive to variability and hence increase the differentiability. Besides, the simulations reveal that growing number of linguistic descriptors can improve both quality of ranking and evaluation accuracy. However, on the other hand, it is harder for users to make precise judgments when the number of descriptors is larger.

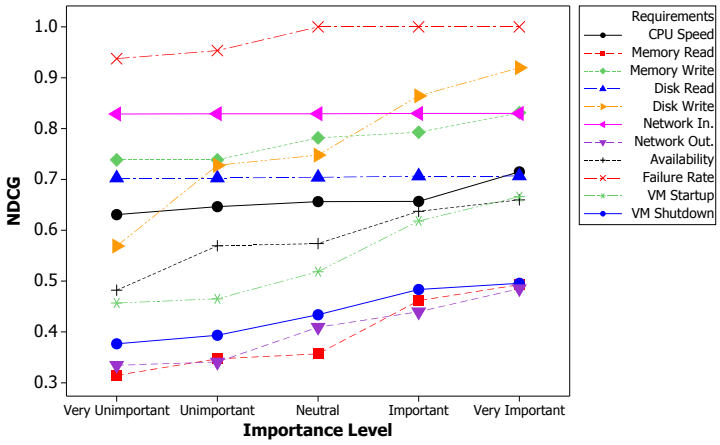
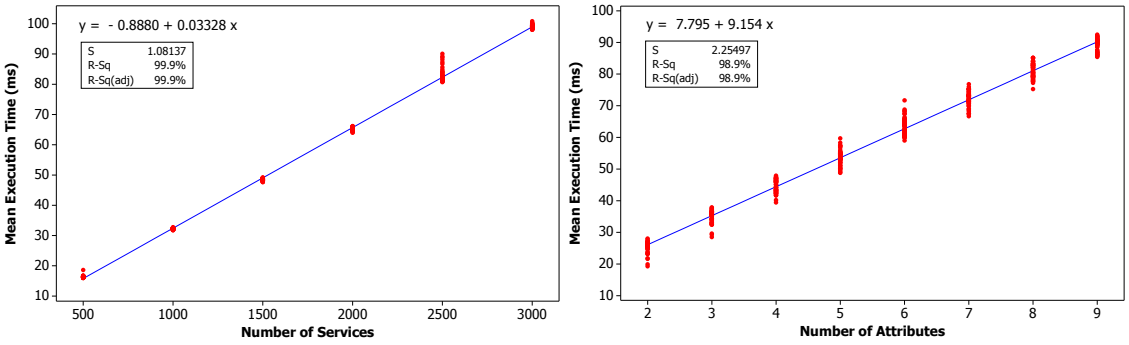


Figure 3.6: Effect of different importance levels on the ranking of services

3.5.3 Effectiveness of Preference Modelling

In the second experiment, we demonstrate the effectiveness of our preference modeling approach. For a query where all its requirements are submitted with *Neutral* importance level, we select one requirement a time and change its importance level to construct new queries. We also use NDCG at position ten as metric, but the relevance scores utilized in this experiment are the trust values of the selected requirement with *Neutral* importance level. In this way, we show whether our approach ranks services that have lower performance deficiencies to the selected requirement higher when the importance level of the requirement increases.

According to the results shown in Figure 3.6, average NDCG remains unchanged for some requirements when their importance level goes up, and the scale of NDCG increase for other requirements also varies, because the space for trade-off is limited. For example, though one Cloud can fully satisfy requirement A, as long as its performance deficiencies to other requirements are unacceptable, our approach considers it unsatisfactory no matter how the user increases A's importance level, which ensures trade-offs are only made within an acceptable level and returned services can reasonably satisfy all requirements.



(a) Mean Execution Time vs Number of Services (b) Mean Execution Time vs Number of Attributes

Figure 3.7: Mean execution time with different numbers of services and attributes

Table 3.3: Cloud Services

Service	CPU Core	Memory(Gb)	Disk(Gb)
A	1	1.7	20
B	1	2.0	18
C	1	1.8	25

3.5.4 Scalability

The third experiment tests the scalability of our approach. We first measure the mean execution time of 25 queries each with requirements for 11 attributes with the number of services increasing from 500 to 3000. After that, we test the average time spent to evaluate 25 queries for 3000 services with the number of leaf attributes considered increasing from 2 to 9. We run 55 tests for each number of services and attributes on a PC with Intel i7-2600 CPU and 4 Gb RAM using a single thread.

As observed in Figure 3.7, our approach is timely efficient and linearly scalable. The execution time can be further reduced if we parallelize the evaluation process.

3.6 Case Studies

In this section, we demonstrate how users can make use of our approach. Table 3.3 shows the basic information of the three example Cloud services involved in the illustrations.

3.6.1 Case 1

As mentioned, apart from web applications, our approach can help other types of applications to identify satisfactory Cloud services, including scientific simulations. In this example, a user who works for a pharmaceutical company wants to run a Monte Carlo simulation in Cloud. For functional requirements, he requires the service to have one core, more than 1.5 Gb RAM, and 18 Gb secondary storage. Though he possesses limited knowledge about computer science, he knows the program is CPU and data-intensive but requires little network communication. Hence he submits *High* requirements for the attributes related to CPU, memory and disk performances and *Low* for the network attributes.

Our method filters service B at discovery phase as it can not provide enough storage. Then it retrieves benchmark traces of A and B for the past few hours to evaluate trust of their current performances. The trust values obtained are 0.85 for A and 0.91 for B, which are all acceptable to the user.

3.6.2 Case 2

A company develops an e-commerce website and plans to deploy it on Cloud. They require their service availability to be at least 99.999%. Through preliminary experiments, they estimate the application requires 7 MIPS CPU and 0.8Mb/s inbound and outbound data transfer speed for a single instance to support 50 requests in parallel. They also identified the network as the bottleneck to their application performance; therefore they raised the importance level of the network to *Very Important*. Their application is undemanding for the disk as it depends on the database deployed on local servers, so they ignore disk I/O attributes. However, they lack the tool to determine the thresholds of memory I/O accurately. Therefore, they choose to submit linguistic requirements for memory I/O. They are aware that there are many sequential memory read and write operations in their program. Thus, they submitted *High* requirements for both memory read and write speed.

Our approach retrieves traces regarding A, B,and C. The traces, in this case, contains

benchmark results conducted on A, B, and C in the previous month as the application is supposed to run for a long time. The returned trust levels are 0.41 for A, 0.92 for B, and 0.9 for C. Both B and C are considered satisfactory to the user requirements.

3.7 Limitations

Like any other service discovery approaches, our method aims to guide users to choose the services that have higher satisfaction possibility, but it is not possible to guarantee the user would receive the services as expected. The accuracy further decreases when the user submit vague linguistic requirements, which is the cost of improving usability.

3.8 Summary

In this chapter, we proposed an efficient trust evaluation approach using hierarchical fuzzy inference system for service selection. The contributions of our method are: 1) it enables trust evaluation of Clouds according to diverse user requirements, 2) it eases the discovery process for both inexperienced and expert users by modeling their vague requirements and uncertain preferences with linguistic descriptors and hedges, and 3) it considers performance variations in the Cloud service discovery phase.

The discovered Cloud services that are satisfactory to user requirements become the candidates for hosting the applications. In the next chapter, we propose a technique that selects a set of Cloud services among the identified candidates considering the application's Service Level Objective (SLO) and operational cost.

Chapter 4

SLO-Aware Deployment of Web Applications Requiring Strong Consistency using Multiple Clouds

To allow web applications requiring strong consistency to be deployed in multiple Clouds, industry and academia have developed various scalable database systems that can guarantee strong inter-data center consistency with reduced network overhead. For applications using these database systems, it is essential to take both network latencies to end users and communication overhead caused by maintaining database consistency into account when selecting the hosting data centers. In this chapter, we propose a technique that identifies satisfactory deployment plan (hosting data centers and request routing), which balances Service Level Objective (SLO) violations, migration cost, and operational cost, for applications requiring strong inter-data center consistency under dynamic workloads. We illustrate how our approach works for applications respectively using two different databases (Cassandra and Galera Cluster), and demonstrate the effectiveness of our approach through simulation studies using settings of two example applications (TPC-W and Twissandra).

4.1 Introduction

WHEN web applications are deployed in geographically dispersed Cloud data centers, application providers need to handle data consistency across data centers (inter-data center consistency), which is challenging for some applications requesting strong consistency, e.g., e-commerce and banking. To make things worse, traditional

This chapter is derived from: **Chenhao Qu**, Rodrigo N. Calheiros, and Rajkumar Buyya, "SLO-aware Deployment of Web Applications Requiring Strong Consistency using Multiple Clouds", *Proceedings of the 8th IEEE International Conference on Cloud Computing (Cloud 2015, IEEE CS Press, USA)*, New York, USA, June 27 - July 2, 2015.

inter-data center commit (two-phase commit) involves high network cost. Therefore, applications often have to adopt eventual consistency (asynchronous replication) to minimize user perceived latencies, which complicates application logic and forces application developers to handle the conflicts and errors caused by inconsistent data [175], even though such cases are rare in production as data synchronization usually completes in a short time in eventual-consistent databases [19].

After realizing that lack of strong consistency has impaired developing productivity, industry and academia shift to developing new databases that can guarantee strong inter-data center consistency [17, 20, 45, 46, 121, 136, 175, 193] to help relieve the programmers' coding burden. Though inter-data center consistency protocols of these new databases are often optimized regarding network overhead, the resulted network delays are still significant and cannot be ignored. Thus, to minimize user perceived response time, it is essential to take the database network delay into account when selecting hosting data centers and when routing requests submitted by different users to the chosen data centers.

In this chapter, we aim to minimize the total excess response time users may perceive beyond the Service Level Objective (SLO) for applications with various inter-data center consistency requirements. The proposed approach benefits application providers so that they can ease their development by adopting these new databases, and in the meantime keep the performance penalties as low as possible.

The contributions of the chapter are two folds. Firstly, we propose a genetic algorithm (GA) that searches a deployment plan (set of data centers and request routing) with a minimum amount of SLO violations when the application is initially migrated to the Cloud. After the initial deployment, the application performance may degrade as time passes due to changes in workload distribution. Secondly, to react to these changes, we propose a decision-making algorithm that continuously optimizes the deployment to balance application performance, redeployment cost, and operational cost. We exemplify how our approach can be applied to two widely used databases (Cassandra [17] and Galera Cluster [45]). To demonstrate the effectiveness of our approach, we conduct simulation studies using settings of two real applications (TPC-W [195], an e-commerce

website, and Twissandra [196], a Twitter-like social network application).

The rest of the chapter is organized as follows. Section 4.2 briefly surveys the existing protocols and databases with strong inter-data center consistency support. Then we describe the target applications and their deployment model in Section 4.3. Section 4.4 explains our approach followed by the performance evaluation in Section 4.5. After that, we discuss some key issues and pitfalls when extending and using our approach in Section 4.6. Finally, we present the related work and summarize the chapter.

4.2 Background

4.2.1 Consistency Protocols

Two-phase Commit

Two-phase commit is the simplest protocol that implements inter-data center transaction commit. Its basic idea is to use one message round to reach a consensus on whether to commit or rollback among all the participating processes and use another round trip to confirm the action with a central coordinator. It is implemented in many distributed databases, such as Google Spanner [46].

Quorum-based Protocols

Quorum-based protocols are used to manage data replication. When writing an object, the system writes to a set of object replicas, called a write quorum. When reading the object, the system fetches it from possibly another set of replicas, called a read quorum. Strong consistency of an object can be guaranteed if the summation of its read and write quorum is larger than the total number of replicas. Some quorum databases [17] also allow users to sacrifice consistency for availability and performance by setting a weaker quorum [19]. However, quorum-based protocols alone are not able to support ACID (Atomicity, Consistency, Isolation, Durability) transactions involving multiple objects.

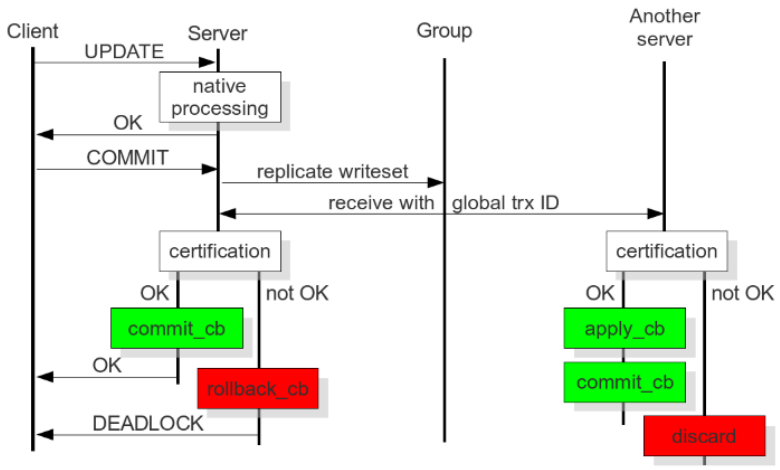


Figure 4.1: Certification-based Commit in Galera [44]

Paxos-based Protocols

Paxos [124] is a family of protocols for reaching consensus in an unreliable distributed environment. In database systems, the most common configuration for the protocol is multi-Paxos [34] with each process act as proposer, acceptor, and learner. The Paxos protocol proceeds in rounds. Its basic implementation also involves two steps, a prepare phase and an accept phase, in the successful case.

In database systems, optimized Paxos protocols are commonly combined with two-phase commit to achieve inter-data center transaction commit. A number of inter-data center transaction commit protocols are built upon Paxos, e.g., MegaStore [20], Spanner [46], MDCC [121], Calvin [193], and Replicated Commit [136].

Certification-based Commit

Certification-based commit [44] is a synchronous replication protocol developed based on the works by Pedone [154], and Kemme and Alonso [115]. Figure 4.1 shows its sequence diagram. The protocol needs the help of an underlying group communication system to deliver the commit requests originated from distributed processes in total and causal order to each process. When doing write transaction, the request is optimistically executed until commit point. After that, the process sends the write change to the whole communication group. The group then returns a global transaction ID to every process.

Since all requests are delivered in the same order, each process can deterministically and independently check potential conflicts in its commit queue using a certification test. The request that passes the test can return immediately.

4.2.2 Databases Supporting Strong Inter-data Center Consistency

Google's Systems

Google have been developing distributed databases that are both highly scalable and strongly consistent. Their first achievement is MegaStore [20]. It implements ACID semantics within each entity group (objects stored together) using synchronous replication based on optimized Paxos, and transaction across entity groups using two-phase commit. The second outcome is Spanner [46], which further supports external consistency (linearizability) with the help of physically synchronized clocks (GPS and atomic clock). Upon Spanner, Google built F1 [175], a distributed relational database system for their critical AdWords platform. It provides more enriched transaction semantics with high availability and scalability. All Google's systems remain proprietary.

Open Source Databases

Cassandra [17] is a shared nothing NoSQL database using quorum-based protocol for its consistency model. It allows users to set individual read and write quorums at the granularity of query. It also provides limited transaction support (lightweight-transaction) starting from version 2.0 using a heavy-weight Paxos consensus protocol, which requires four round-trip messages to complete.

Galera cluster [45] is an open source scalable synchronous replication solution developed and maintained by Codership for MySql. Galera's replication is based on certification-based commit [44] shown in Figure 4.1. Since replication is synchronous, read-only queries in Galera are always processed locally.

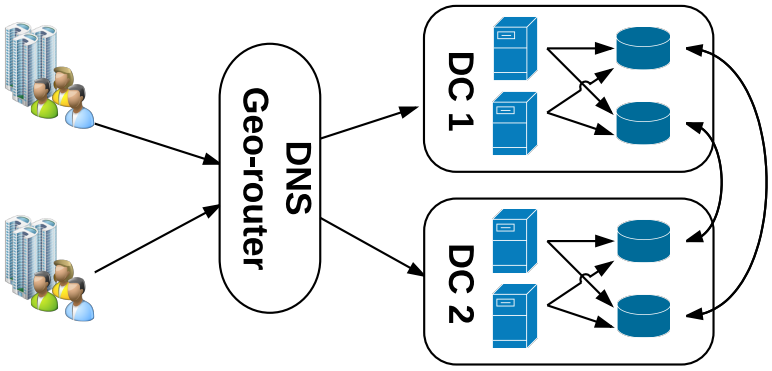


Figure 4.2: Deployment using 2 data centers

4.3 Application and Deployment Model

4.3.1 Target Applications

We target session-based web applications. We assume the delay of the application is dominated by the round-trip time (RTT) between different parties, as the processing time of the request can be considered constant provided that enough computing resources are provisioned. In this case, whether the SLOs can be met is largely determined by the involved network latencies.

To benefit from our work, the application should also be deployed in geographically dispersed data centers, and some of its requests should require strong consistency, e.g., a group working application that always reflects the newest updates to its end-users, a social-network application that consistently and timely shows people's posts and comments, or a distributed banking application that needs to satisfy ACID semantics.

4.3.2 Deployment Model

We assume the whole software stack of the application (including application servers and underneath databases) is deployed in multiple geographically dispersed data centers and each application replica can autonomously scale up and down according to the changing workloads, as shown in the example in Figure 4.2. We assume all chosen data centers have the full copy of data. Companies, like Facebook [149], commonly adopt this approach. Furthermore, the databases studied in this chapter, Cassandra and Galera

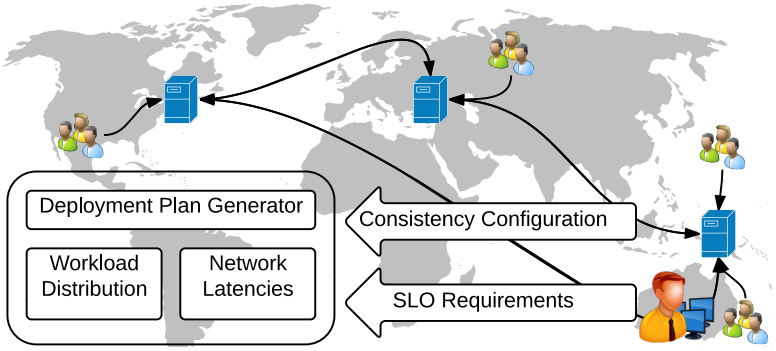


Figure 4.3: The Proposed Approach

Cluster, support only full replication for multi-data center deployment within the same keyspace or namespace. The target applications should also use shared-nothing multi-master database clusters, which means all database queries originated from any server can be served by database nodes collocated in the same data center. Depending on the database and different queries' consistency requirements, we also consider the network delays caused by communications among database cluster nodes located in different data centers into the problem model. All inner-data center communications, otherwise, are omitted.

We classify users into groups according to their geographic locations. All requests from the same location are routed to the same data center using DNS routing services similar to Amazon Route 53's Geo Routing [14].

4.4 Proposed Approach

4.4.1 Overview

Our approach requires application administrator to provide SLO and consistency configuration for each type of request as illustrated in Figure 4.3. Besides, it needs the information of request number coming from each location. These data can be recorded during production. Furthermore, it requires the network latency data between each data center and each location and latency data between each data center. Since it is hard to collect all the real-time RTT latencies between each site and data center given the large number

of them, we can either rely on network predictors, like the one employed by Grozev and Buyya [78], to estimate unknown latencies, or trusted third parties like NetMetrics [151], which is a global Internet performance database, to provide the latency information.

The objective of our work is to select and manage a subset of data centers to host the application replicas, and in the meantime, find the optimal request routing according to the chosen data centers, so that the total amount of estimated SLO[1] violations is as small as possible. The approach involves two steps:

Initial Deployment: When the application is initially migrated to the Clouds, our approach aims to select the hosting data centers and route requests to chosen data centers with minimum amount of total estimated SLO violations according to the current geographical distribution of requests[2].

Deployment Optimization: In the second step, our approach continuously attempts to maintain high performance of the application by contracting, optimizing, or expanding the deployment with acceptable migration[3] efforts in response to changes in the requests distribution.

In this chapter, we use the term **expand** and **contract** respectively for increasing and decreasing the number of chosen data centers. We focus on the geographical distribution of resources instead of the total resource amount, for which the term commonly used is scaling up and down.

4.4.2 SLO Violation Model

We first propose a model to estimate the amounts of SLO violations incurred by specific deployment plans. It is composed of the general model, which views database network latencies as a black box and extracts commonalities of the target applications, and

[1] As we assume the request processing time is constant, the SLO hereafter is referred to as the desirable total network latency.

[2] Distribution of requests refers to the ratio of requests coming from each geographical location.

[3] In this chapter, migration means deployment change that requires moving data to another data center, including moving an existing replica to another data center and deploying a new replica in a data center, but excluding removing an existing replica from a data center.

Table 4.1: Symbols of the General Model

Term	Meaning
$\mathbf{M}$	Set of available data centers
$\mathbf{G}$	Set of geographical locations
$\mathbf{I}$	Set of request types
$\mathbf{X}$	Set of the chosen data centers to host application replicas
H	Number of chosen data centers
N_i^l	Number of type l requests from location i
T^l	Latency SLO of type l request
R_{ij}	RTT latency between location i and data center j
s_i	Total estimated SLO violations at location i given $\mathbf{X}$
lt_{ij}^l	Estimated network latency of type l request at location i if that request is served by the application replica placed in data center j
dlt_j^l	Database network latency for type l request served in data center j given $\mathbf{X}$
p^l	Protocol overhead of type l request (number of RTTs)

the database model, which allows providers to plug different databases into the general model.

The General Model

For clarity, we introduce a metric **Average Violation Per Request (AVPR)**, calculated as the total estimated excess waiting time beyond defined SLOs (SLO violations) for all requests divided by the total number of requests, as the optimization target. Using the symbols in Table 4.1[4], the general model then is defined as:

$$\begin{aligned} &\text{minimize} \quad f_{AVPR}(\mathbf{X}) = \frac{\sum_i s_i}{\sum_i \sum_l N_i^l} \quad \forall i \in \mathbf{G}, l \in \mathbf{I} \\ &\text{subject to} \quad \mathbf{X} \subset \mathbf{M}, |\mathbf{X}| = H \end{aligned}$$

where s_i is the amount of total SLO violations at location i. N_i^l is the number of type l requests coming from location i. $\mathbf{X}$ and $\mathbf{M}$ respectively represents the set of selected data centers, and the set of available data centers. H is the number of chosen data centers. $\mathbf{G}$ is the set of geographical locations. $\mathbf{I}$ is the set of request types.

[4]The symbols representing sets are shown in bold in the following tables and text.

When calculating s_i, we first need to determine the routing of requests from each location i according to the chosen $\mathbf{X}$. The data center within $\mathbf{X}$ that incurs the least amount of SLO violations is selected to serve users at location i. The amount of SLO violations incurred by each data center for serving users at location i is computed as the sum of all the excessive latencies that the users are estimated to perceive beyond SLO. Thus, s_i, in formula, can be represented as:

$$s_i = \min_{j \in \mathbf{X}} (\sum_l N_i^l (lt_{ij}^l - T^l)) \;\; \forall lt_{ij}^l > T^l, l \in \mathbf{I}$$

where T^l is the SLO of type l request. lt_{ij}^l is the latency of type l request perceived by users at location i, if it is served by the replica in data center j. lt_{ij}^l can be estimated using the following formula:

$$lt_{ij}^l = p^l R_{ij} + dlt_j^l \;\; j \in \mathbf{X}, l \in \mathbf{I}$$

lt_{ij}^l is calculated as the sum of two parts. The first part is the network latency between the user location i and the corresponding serving data center j ($p^l R_{ij}$). p^l is the overhead (number of RTTs) of the communication protocol used by the type l request. R_{ij} is the RTT latency between location i and data center j. The second part is the database network latency overhead (dlt_j^l) introduced below.

The Database Model

Modeling of database network latency overhead is database-specific. In this section, we illustrate how to model the overhead of two widely-used databases. One is Cassandra [17], a NoSQL database; the other is Galera Cluster [45], a replication solution for MySql relational database.

The Cassandra Model: The widely-adopted replication strategy for Cassandra involving multiple data centers in production is symmetric replication, where each data center stores the same number of replicas [17]. Cassandra uses quorum-based protocol to implement consistent read/write operations across replicas, and it supports various consistency configurations at the granularity of query. Given the set of selected data centers

Table 4.2: Symbols of the Cassandra Model

Term	Meaning
$\mathbf{R}_l$	Set of read queries in type l request
$\mathbf{W}_l$	Set of write queries in type l request
Qr^l_k	Read quorum of the k_{th} read query in request type l
Qw^l_m	Write quorum of the m_{th} write query in request type l
r	The replication factor of data centers
$\alpha(j,k,\mathbf{X})$	The function finds the k_{th} shortest RTT latency among all latencies between each data center within $\mathbf{X}$ and data center j

($\mathbf{X}$), using the symbols in Table 4.2, its database network overhead dlt^l_j can be modelled as:

$$dlt^l_j = \sum_k \alpha(j, \lceil \frac{Qr^l_k}{r} \rceil, \mathbf{X}) + \sum_m \alpha(j, \lceil \frac{Qw^l_m}{r} \rceil, \mathbf{X}) \quad \forall k \in \mathbf{R}_l, m \in \mathbf{W}_l \\ j \in \mathbf{X}, l \in \mathbf{I}$$

where $\mathbf{R}_l$ ($\mathbf{W}_l$) is the set of read (write) queries in type l request, and Qr^l_k (Qw^l_m) is the read (write) quorum of the k_{th} (m_{th}) read (write) query in type l request. r is the replication factor in a data center. The function $\alpha(j,k,\mathbf{X})$ returns the k_{th} shortest RTT latency among all latencies between each data center within $\mathbf{X}$ and data center j. Following the work by Shankaranarayanan et al. [171], we model the delay of the read/write query as the slowest replica's response time in the quorum. For example, if the read quorum is 3 and each data center holds 1 data replica, Cassandra will wait to receive replies from the 2 replicas located in other data centers as the network delay to the local copy is orders of magnitude smaller. Hence, the resulted delay will normally be the second shortest RTT latency from the local data center j to the other selected data centers.

In Cassandra, the remote replica only replies digest of the objects. If the local copy is stale, it will send another request to fetch the complete data and update all the stale replicas. Similar to Shankaranarayanan et al. [171], we ignore this overhead as such case is rare and, thus, only impose a little impact on the average delay of all the requests.

The administrator is responsible for deciding the quorum settings of each query, as, besides consistency and performance, other concerns in this process may complicate the decision, such as availability ($Qr = 1, Qw = H$ maximizes the performance for read-

Table 4.3: Symbols of the Galera Model

Term	Meaning
$\beta(j, \mathbf{X})$	The function finds the largest RTT latency among all latencies between each data center within X and data center j
V^l	Number of transactions that have write operations in request type l

intensive applications but is susceptible to failures). For query requiring strong consistency, application administrator should specify its read/write quorum so that the object's Qr and Qw satisfy $Qr + Qw > H$. Certainly, it is also possible to set a weaker configuration [19] if strong consistency is unnecessary.

In Cassandra, the legitimate quorum settings are currently limited to "**ONE**, **TWO**, **THREE**, **ALL**, and **QUORUM** (simple majority)". Some configurations are invalid, e.g., ($H = 5$, $Qr = 2$, and $Qw = 4$), as $Qw = 4$ is not allowed. However, we also include these configurations in our evaluations as we suppose they will be supported by future versions.

The Galera Model: In Galera cluster, all read-only transactions are executed locally while transactions with write operations synchronously replicated to all remote replicas using certification-based commit [44]. As shown in Figure 4.1, there is no further group communication involved in the protocol [44] after the transaction ID is determined, the network latency is dominated by the data center that has the largest RTT latency to the request originator. Based on symbols in Table 4.3, dlt^l_j, in this case, can be simply formulated as:

$$dlt^l_j = V^l \beta(j, \mathbf{X}) \;\; j \in \mathbf{X}, l \in \mathbf{I}$$

where V^l is the number of database transactions that have write operations in type l request, and the function $\beta(j, \mathbf{X})$ returns the largest RTT latency among all latencies between each data center within $\mathbf{X}$ and data center j.

Galera nodes may queue the messages before delivering them due to the group communication overhead. We neglect this delay because we believe it is unpredictable, application-specific, and also insignificant compared to the network transfer delay. To build a more

precise model, application administrators can profile their applications to obtain the average queuing time and add this value to the model.

4.4.3 Solution for Initial Deployment

Hardness of the problem

The problem of moving several application replicas to the Cloud falls in the category of Facility Location Problems [174], which are usually NP-hard to solve.

Proving by restriction, which aims to demonstrate the target problem contains an already-known NP-hard problem as a special case, is a common method to prove a problem is NP-hard. Hereafter, we prove our problem is NP-hard by showing that the NP-hard k-median problem is a special case of it.

Proof: Suppose eventual consistency is used by all the request types, then dlt_j^l equals 0 for all j, l. Therefore, $lt_{ij}^l = \min_{j \in \mathbf{X}}(p^l R_{ij})$, which is constant. With fixed amount of estimated SLO violations between any location i and any candidate data center j, selecting H data centers from a set of candidate data centers $\mathbf{M}$ to serve customers at a set of locations $\mathbf{G}$ with minimum amount of SLO violations is exactly the k-median problem.

Since our problem is NP-hard, we refer to heuristic approximation algorithms that can find good enough solutions in polynomial time.

Genetic Algorithm Overview

Our solution is based on genetic algorithm (GA). Compared with other approaches, it has three advantages. The first is that meta-heuristics like GA are more flexible. For each database, the administrator only needs to substitute the representation of dlt_j^l to let the algorithm work. While for other heuristics, such as greedy algorithms, we find that they are often tightly coupled with the database models. The second is that GA produces satisfactory results in our context. We demonstrate that in our experiments. The last but not least is that it is easy for meta-heuristic algorithms like GA to incorporate other selection criteria into the existing model, e.g., the number of migrations.

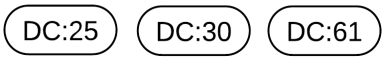


Figure 4.4: An Example of the Chromosome with 3 chosen data centers

Genetic Algorithm in Detail

Our GA generates a set of random solutions at the beginning and then iteratively performs crossover, mutation, and selection according to a predefined fitness function. It returns not only the set of chosen data centers but also the optimal request routing regarding the selected data centers. Before calculating fitness value, it respectively picks the optimal data center among the chosen data centers **X** to process the requests from each user location. The fitness function of the algorithm is defined as the f_{AVPR} function in the SLO violation model.

GA requires programmers to encode the solutions using a particular data structure, called chromosome. In our GA, we number all the available data centers and encode the solution as a non-repetitive ascending array, as illustrated in Figure 4.4. The number of genes in a chromosome equals the total number of data centers the provider wants to choose H. Thus, repetitive genes are not allowed in the chromosome. Besides, we sort the genes in ascending order for the convenience of programming.

We wrote our initial population generator, crossover, and mutation operators. For **mutation operation**, it first randomly picks one gene in the chromosome. Then it mutates value of the gene. The mutated gene should be unique to all the genes in the previous chromosome. Finally, the new chromosome is sorted to preserve the ascending property. The **initial population** is generated randomly. The genes in an initial chromosome are generated stochastically one by one unique to the previously created genes on the same chromosome. After that, the genes are sorted by the ascending order. For **crossover operation**, we randomly swap some genes of the two randomly chosen chromosomes. If the resulted new chromosomes have repetitive genes, we perform additional mutations to eliminate repetitions. Then the resulted new genes are sorted to the ascending order. The algorithm terminates if not enough progress has been made for some time.

Table 4.4: Symbols of Deployment Optimization

Term	Meaning
$n\_migration$	The function calculates the number of required migrations
t	Redeployment interval
U	Upper bound of $AVPR$
L	Lower bound of $AVPR$
W	Unit AVPR gain threshold for migration if $AVPR$ of the current deployment is below U
C	Cooling period for contracting the application

4.4.4 Solution for Deployment Optimization

Decision-making Algorithm

In this step, we aim to optimize the deployment according to the given workload and the current deployment. We propose a decision-making algorithm (Algorithm 1) to decide whether and how to adjust the deployment so that satisfactory enough $AVPR$ can be achieved with acceptable migration efforts and operational cost. In our solution, we require the administrator to specify the upper threshold of the acceptable $AVPR$, represented as U, and the lower threshold of the $AVPR$, which is shown as L.

The algorithm uses redeployment heuristics to find the satisfactory redeployment plan. Their objective is to let providers gain more $AVPR$ improvement from unit migration effort, which is defined as:

$$\begin{aligned} max \quad & \frac{U - f_{AVPR}(new)}{n\_migration(new, current)} && f_{AVPR}(current) \geq U \;|| \\ & && new.size < old.size \\ max \quad & \frac{f_{AVPR}(current) - f_{AVPR}(new)}{n\_migration(new, current)} && otherwise \end{aligned}$$

where $n\_migration(new, current)$ returns the number of migrations required to change the current deployment to the new deployment. The above function means when current deployment cannot meet the upper bound of $AVPR$, or the algorithm is trying to contract the application, the optimization target is to maximize the $AVPR$ reduction from unit migration effort against the upper bound. Otherwise, it is to maximize the $AVPR$

Algorithm 1: Redeployment Decision-making Algorithm

```
   Input: initial_dc_num, and t
1  dc_num = initial_dc_num;
2  current_plan = first_step_deployment(dc_num);
3  for every t do
       /* try to contract the application                          */
4      if f_AVPR(current_plan) < L for consecutively more than C rounds then
5          new_plan;
6          for each dc ∈ current_plan do
7              contracted_plan = current_plan.remove(dc);
8              tmp_plan = redeployHeuristic(dc_num − 1, contracted_plan);
9              if tmp_plan.isFitter(new_plan) then
10                 new_plan = tmp_plan;
11             end
12         end
13         if f_AVPR(new_plan) < U then
14             current_plan = new_plan;
15             dc_num − −;
16             continue;
17         end
18     end
       /* try to optimize the deployment with the same number of
          chosen data centers                                      */
19     new_plan = redeployHeuristic(dc_num, current_plan);
20     if worthwhile(current_plan, new_plan) then
21         current_plan = new_plan;
22         continue;
23     end
       /* expand the application                                   */
24     if f_AVPR(new_plan) ≥ U then
25         new_plan = redeployHeuristic(++dc_num, current_plan);
26         current_plan = new_plan;
27     end
28 end
```

reduction against the $AVPR$ of the current deployment.

As noted in Algorithm 1, our approach first searches if there is a chance to contract the application when $AVPR$ has been below the lower bound L for a time longer than the

Algorithm 2: Worthwhile Method

Input: *current_plan*, and *new_plan*

```
1 if f_AVPR(current_plan) < U && w(new_plan) > W then
2 |   return true;
3 end
4 if f_AVPR(current_plan) ≥ U && f_AVPR(new_plan) < U then
5 |   return true;
6 end
7 return false;
```

cooling period C (Line 4-18). We introduce the cooling time here to alleviate oscillation that would cause frequent contraction and expansion of the application. As removing one data center is not counted in the migration effort, when contracting the application, the algorithm iterates all the possible choices and tries to find the best redeployment plan. If no contraction is performed, it then endeavors to find a better deployment with the same number of chosen data centers (Line 19-23). Redeployment will only be conducted if $AVPR$ reduction from unit migration effort is beyond the administrator-defined threshold W or the new deployment can reduce the $AVPR$ to the acceptable level (Algorithm 2). Suppose no deployment plan that will reduce the $AVPR$ into the acceptable level is found, the algorithm then expands the application (Line 24-28).

Administrators can reset constants U, L, C, and W at any time according to their needs.

Redeployment Heuristics

We developed two redeployment heuristics:

Migration-aware Genetic: It replaces the fitness function of the GA used in the initial deployment phase with the migration-aware optimization target.

***k*-Brute Force:** Brute-forcedly search the best plan that is reachable using at most k migrations (feasible for small k if H, $|\mathbf{G}|$, and $|\mathbf{M}|$ are large). In our experiments, we set k to 2.

4.5 Performance Evaluation

We evaluate our approach using simulations. Settings of the data centers and networks are described in the next sub-section. Workloads and baselines used are explained within each experiment. As GA is stochastic, we run each test 5 times and report the best result which is chosen to guide the deployment. For parameters of the GA, we set the population size to 1000, the crossover rate to 50%, and the mutation rate to 2%. We select half of the best chromosomes for reproduction after each iteration.

4.5.1 Data Centers and User Settings

We use the data collected by Zhu et al. [227] for our experiments due to lack availability of latency data from real Cloud data centers. The dataset uses 307 PlanetLab nodes as the candidate data centers and 1881 web services discovered by a crawler as the user locations. Zhu et al. let the PlatetLab nodes ping the web services and each other to obtain the real RTT latency data. Though PlanetLab nodes are not commercial Cloud data centers, we believe the dataset is still representative of our problems as they are geographically distributed and can be viewed as mimics of Cloud data centers in which application replicas interact with each other and end users through the Internet.

4.5.2 Evaluation of Initial Deployment

Workload

We studied two real-world applications and specified the consistency requirements of all their request types according to our judgement. The first application is the TPC-W workload [195] which mimics an e-business website. The TPC-W implementation we studied uses MySql database, which is compatible to the Galera cluster. The second application is called Twissandra [196], which is an open source copy of Twitter built upon Cassandra.

We generated two different workloads for both applications from each geographical location using the normal distribution. The request mix of TPC-W was defined by the browsing and ordering workload included in its benchmark suite. Roughly, in browsing

workload, 75% of the requests could be served without inter-data center communication; while in ordering workload, the number decreased to about 38%. For Twissandra, the request mix is the ratio of timeline view and tweet operations. For read-intensive workload, the timeline view/tweet ratio was set to 9:1; for write-intensive workload, it was 7:3. We assumed strong consistency is required for both operations, which means Qr(timeline view)+Qw(tweet)$> H$. We set $Qr = 2, Qw = H - 1$ for all Twissandra tests using multiple Clouds, as Twissandra is relatively read-intensive, and $Qr = 2$ can tolerate one data center down when $Qw > 1$ for $Qr + Qw - 1$ data centers.

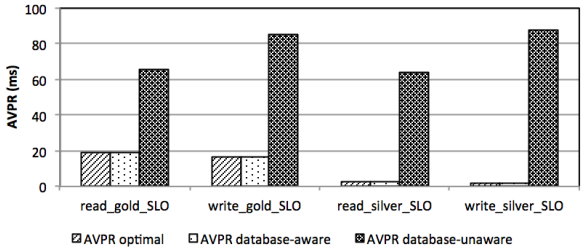
SLO

We specified different latency SLOs as well. The latency constraint for each request type was set according to the total network round-trips it requires. Each request type was respectively given $50ms$ and $100ms$ to perform one network round-trip for **Gold** and **Silver** SLO.

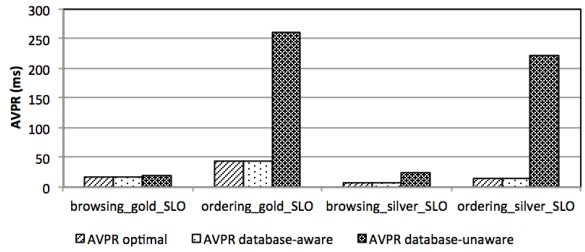
Necessity of Considering Database Network Latencies

First, we show that it is important to consider database network latencies when deploying applications requiring strong inter-data center consistency. We fixed the number of chosen data centers to 3 in the experiment, and run our consistency-aware GA algorithm with different levels of SLO and workloads. Results are compared with a baseline GA algorithm that only considers network communications between data centers and end users, which is similar to the setting used in Kang et al.'s work [111]. We run the baseline and then evaluated the found solutions using the database latency-aware model.

From Figure 4.5, it is evident that by omitting database network latencies, the found solutions resulted in significantly higher $AVPR$ compared to the database consistency overhead-aware approach, except cases of TPC-W application under the browsing workload. The produced differences of the two algorithms for TPC-W under browsing workload are much smaller because the majority of requests (75%) in browsing workload are served without inter-data center communication. From the results, we can conclude that



(a) $AVPR$ for Twissandra using various SLOs and workloads



(b) $AVPR$ for TPC-W using different SLOs and workloads

Figure 4.5: Comparing performances of different algorithms using 3 data centers

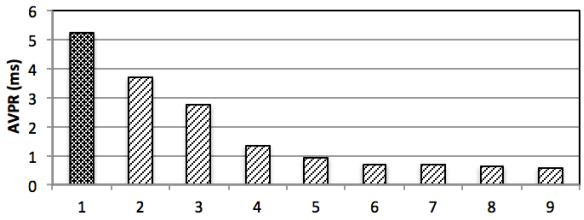
it is important to take database network latencies into account if the inter-data center communication rate is relatively high in the request mix.

Efficacy of our GA

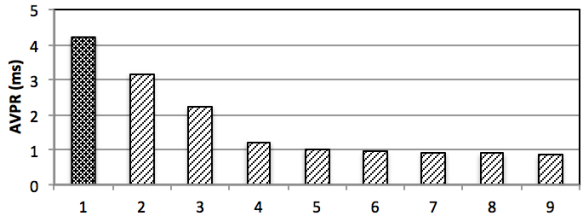
We compared the results of our GA with the optimal results. Optimal solutions were derived by traversing all the possible solutions in the search space. In our settings, finding optimal results using 3 data centers was the limit on our desktop testbed, which took more than 16 hours to finish. As shown in figure 4.6, our GA found the exact optimal deployment plan for all eight settings but using only around 4 minutes.

Efficacy of Deploying Applications in Multiple Data Centers

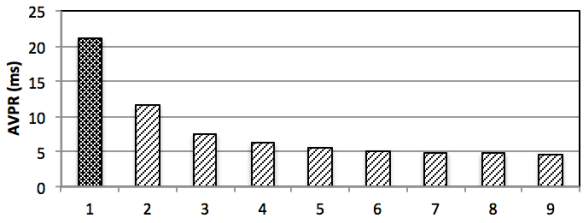
Regarding $AVPR$, we discuss whether it is worth to deploy applications requiring strong inter-data center consistencies in multiple data centers. In this experiment, we fixed the SLO to Silver and ran our GA algorithms with different numbers of data centers and



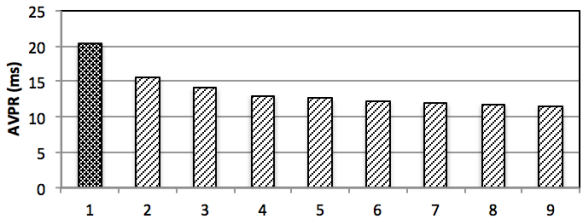
(a) $AVPR$ for Twissandra read-intensive workload



(b) $AVPR$ for Twissandra write-intensive workload



(c) $AVPR$ for TPC-W browsing workload



(d) $AVPR$ for TPC-W ordering workload

Figure 4.6: Comparing deployments using multiple data centers and optimal deployments using 1 data center

workloads. The results are compared with the optimal deployments using one data center only.

The results presented in Figure 4.6 indicate that, purely from the performance perspective, deploying applications requiring strong consistency in multiple data centers is still beneficial. Using 3 data centers can reduce the amount of SLO violations to half of that using optimal single data center deployment. However, the performance gain from increased number of data centers becomes negligible when the number of data centers

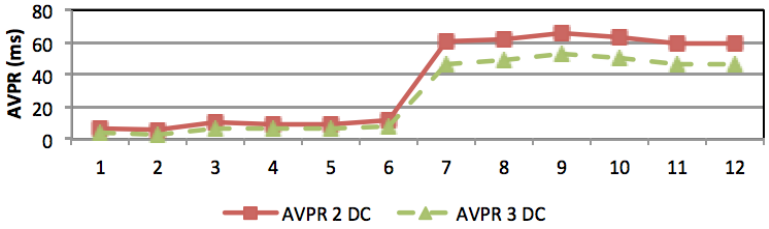


Figure 4.7: Results using fixed deployments and Silver SLO

exceeds 4. This phenomenon justifies our motivation to keep the number of chosen data centers as small as possible in the deployment optimization phase to save operational cost. Even though the performance gain is small, the providers may still want to deploy their applications in a larger number of data centers for other benefits, such as availability and fault-tolerance, which is out of the scope of this chapter.

4.5.3 Evaluation of Deployment Optimization

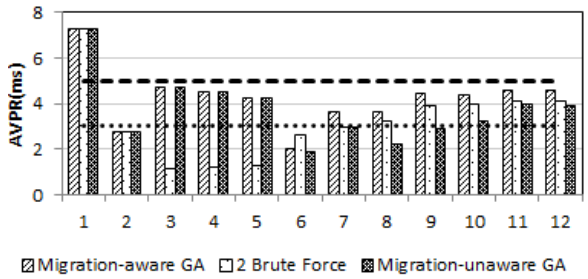
Workload

We generated a series of workloads for Twissandra to simulate the expansion of business and workload increase for the test of our redeployment decision-making algorithm. We classified the user locations into 11 geographic categories based on their latencies to all the 307 data centers using K-means, and we added one category of locations into the workload per redeployment round. The numbers of requests at the added locations continuously grew in the following rounds in our settings to mimic workload increase.

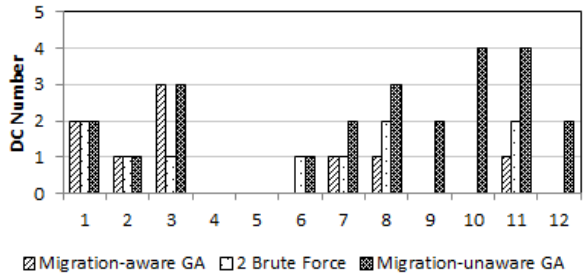
Necessity of Deployment Optimization

To illustrate the necessity of deployment optimization along with the business expansion and workload increase, we ran experiments using 2, and 3 data centers according to the initial workloads and observed how the performances of the fixed deployments changed in the later times under varied workload distributions.

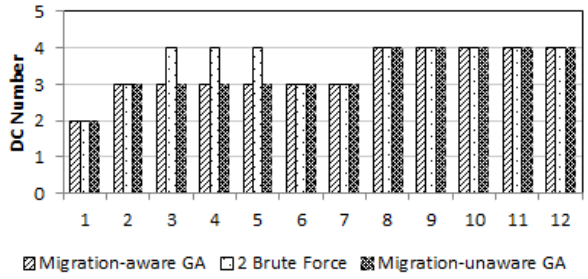
As Figure 4.7 shows, the fixed deployments incurred unacceptably high $AVPR$ when the workload has been expanded and increased, which indicates that redeployment is



(a) AVPR of Deployments



(b) Number of Migrations



(c) Number of Chosen Data Centers

Figure 4.8: Results with $initial\_dc = 2$, $U = 5$, $L = 3$, $W = 0.5$, $C = 1$, and Silver SLO

essential to maintain acceptable QoS under changing workloads.

Our Approach

We tested our decision-making algorithm using the two proposed redeployment heuristics. We compared the results with one baseline algorithm.

Similar to Algorithm 1, the baseline algorithm tries to contract the application when $AVPR$ has been below the lower bound L for the time longer than the cooling period and expands the application when U cannot be met. However, it performs migrations as long as it finds a better deployment plan, no matter whether the improvement is significant

or not compared to the current deployment. The redeployment heuristic used in the baseline is called Migration-unaware Genetic. Its target is always to find the deployment with minimum $AVPR$, which is the same GA utilized in the initial deployment phase.

We ran the test with $initial\_dc = 2$, $U = 5$, $L = 3$, $W = 0.5$, $C = 1$, and Silver SLO. Figure 4.8 shows the comparison of $AVPR$, number of migrations, and number of chosen data centers in each redeployment round using the proposed and the baseline approaches. Except for round 3-5, the baseline algorithm found the deployment plans with the smallest $AVPR$. On the other hand, it incurred the largest number of migrations. Our approaches, though resulted in a little more $AVPR$, managed to maintain the performance under the upper bound using significantly fewer numbers of migrations (9:10:24), which shows the effectiveness of our decision-making algorithm in balancing $AVPR$ and the migration cost.

The 2-brute force heuristic failed to find a redeployment plan using three data centers due to its limitation that, within each round, maximum two migrations can be conducted, comparing to the migration-aware GA heuristic, at the 3rd round. Though it significantly outperforms other approaches in $AVPR$ during round 3 to round 5, it uses one more data center, which increased the operational cost. The 2-brute force approach finally took the chance to contract the application at round 6 after long passing the cooling period due to its limitation which caused it not being able to find a redeployment plan with $AVPR$ below the upper bound using three data centers during round 4 to round 5.

Choosing the right redeployment heuristic is always context specific. If the workload distribution does not change abruptly, k-brute force is the better choice, as it provides a higher chance for the providers to find the redeployment plan with optimal $AVPR$ reduction from unit migration. For application providers that have tight operational budget or providers that are expanding quickly, then migration-aware GA is possibly the right choice.

4.6 Discussions

Our approach can be applied to applications using other database systems, as long as they adopt shared-nothing architecture, store full copy of data at each site, and employ a known inter-data center consistency protocol. Though, currently, not many commercial databases support inter-data center consistency, some emerging ones satisfy the prerequisites and are compatible with our approach, such as MDCC [121], Calvin [193], and Replicated Commit [136].

Furthermore, it is difficult to build an accurate database latency model that takes all the cases into consideration. Like what we have done with the Cassandra and Galera model, we believe a close approximation is enough to meet the purpose as the rare cases only have a small influence on the aggregated results.

Our approach aims to provide a performance boost in long-term. Handling performance issues caused by short-term network instability is out of our scope, and there is no solution can realize that if application cannot be migrated in short time. The network data used should be representative of their usual performance to obtain satisfactory deployment.

Providers can either use the current workload or the predicted workload to generate the redeployment plan. If predictions can be made accurately, using predicted workload can further improve the performance by preparing for the workload changes during the redeployment intervals in advance.

4.7 Related Work

Previous works about inter-data center consistency mostly focus on the database layer. Besides developing databases that are capable of supporting strong inter-data center consistency [17, 20, 45, 46, 121, 136, 175, 193], some works have also explored optimizing the placement of data replicas and their consistency configurations to reduce the response time and cost in the database layer. These works usually target quorum-based systems, as they are more flexible in adjusting consistency configurations. SPANStore [213] is a multi-data center key-value store with quorum consistency. It can transparently place

data replicas across geo-distributed data centers so that total cost of storage and I/O operations is minimized, and meanwhile, it still can meet its latency, fault-tolerance, and consistency goals. Shankaranarayanan et al. [171] proposed an approach to find the optimal configuration of quorum-based data across multiple data centers (number of replicas, replica placement, Qr, and Qw) so that read/write latency is minimized in common cases and bounded when one data center is down. Our work is different to theirs as we aim to optimize the performance of the whole application by selecting a set of hosting data centers instead of optimizing placements of data among a set of already selected data centers. Besides, we aim to build a general approach that is extensible to support multiple databases.

Many works have studied the application placement problem in multi-data center context. However, none of them have considered inter-data center consistency. Kang et al. [113] explored how to deploy and redeploy standalone application replicas to minimize total response time or maximize user satisfaction in changing workload. Zhang et al. [224] proposed an approach to dynamically place applications in geographically distributed Cloud data centers with limited capacities and volatile costs using control and game theory. Wu et al. [212] targeted the deployment of social media applications using multiple Clouds. Their approach employed a social influence model to predict the future demand, and then judiciously place the media files and servers in Cloud data centers to achieve minimum cost under latency, bandwidth, and availability constraints.

4.8 Summary

We proposed an approach to help web application providers deploy their applications with various inter-data center consistency requirements across multiple Cloud data centers. It generates deployment plan with a minimum amount of SLO violations when the application is first moved to the Cloud using our genetic algorithm (initial deployment phase), and then it continuously optimizes the deployment considering SLO satisfaction, migration cost, and operational cost along with the change of workload distribution (deployment optimization phase). We proposed an extensible SLO violation model so that

besides the illustrated database systems (Cassandra and Galera Cluster), other databases that satisfy certain requirements can be easily adapted to our approach. To demonstrate the effectiveness of our approach, we conducted simulation experiments using settings of two applications (TPC-W and Twissandra).

After the application is deployed in the Clouds, it comes to the task to ensure just enough resources are provisioned to the application during its life cycle so that the QoS requirements can be met with the minimum cost incurred. In the next chapter, we shift our focus to the provisioning aspect of web application management and propose an auto-scaler for web applications using heterogeneous spot instances.

Chapter 5

A Reliable and Cost-Efficient Auto-Scaling System for Web Applications Using Heterogeneous Spot Instances

Spot instances sold through an auction-like mechanism are usually 90% cheaper than on-demand instances. However, they can be terminated by providers when market prices exceed prices that users bid. Thus, they are considered unreliable and are widely used to provision fault-tolerant applications only. However, in this chapter, we propose a novel auto-scaler using heterogeneous spot instances, which achieves both high availability and significant cost saving. We implemented two prototype systems respectively on a simulation testbed and Amazon EC2. The experiments on both platforms prove the efficacy of our approach.

5.1 Introduction

THERE are three common pricing models in current Infrastructure-as-a-service (IaaS) Cloud providers, namely *on-demand*, in which acquired virtual machines (VMs) are charged periodically with fixed rates, *reservation*, where users pay an amount of up-front fee for each VM to allow cheaper use of the VM within a certain contract period, and *rebated*.

Spot instances, a rebated pricing model, were introduced by Amazon to sell their

This chapter is derived from: **Chenhao Qu**, Rodrigo N. Calheiros, and Rajkumar Buyya, " A Reliable and Cost-Efficient Auto-Scaling System for Web Applications Using Heterogeneous Spot Instances", *Journal of Network and Computer Applications*, issue 65, page 167 - 180, 2016. **Code Available:** https://github.com/quchenhao/spot-auto-scaling

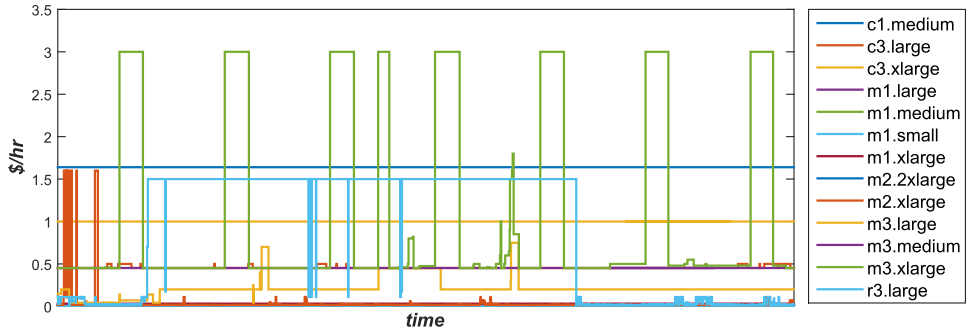


Figure 5.1: One week spot price history from March 2nd 2015 18:00:00 GMT in Amazon EC2's ***us-east-1d*** Availability Zone

spare capacity in the open market through an auction-like mechanism. The provider dynamically sets the market price of each VM type according to real-time demand and supply. To participate in the market, a Cloud user needs to give a bid that specifies the number of instances for the type of VM he wants to acquire and the maximum unit price he is willing to pay. If the bidding price exceeds the current market price, the bid is fulfilled. After getting the required spot VMs, the user only pays the current market prices no matter how much he bids, which results in significant cost saving compared to VMs billed at on-demand prices (usually only 10% to 20% of the latter) [9]. However, obtained spot VMs will be terminated by Cloud provider whenever their market prices rise beyond the bidding prices.

Such model is ideal for fault-tolerant and non-time-critical applications such as scientific computing, big data analytics, and media processing applications. On the other hand, it is believed that availability- and time-critical applications, like web applications, are not suitable to be deployed on spot instances.

Adversely in this chapter, we illustrate that, with the effective fault-tolerant mechanism and carefully designed policies that comply with the fault-tolerant semantics, it is also possible to reliably scale web applications using spot instances to reach both high QoS and significant cost saving.

The spot market is similar to a stock exchange that, though possibly following the general trends, each listed item has its unique market behavior according to its supply and demand. In this kind of market, often price differences appear with some types of

instances sold at high prices due to strong demand, while some remaining underutilized leading to attractive deals. Figure 5.1 depicts a period of Amazon EC2's spot market history. Within this time frame, there were always some spot types sold at discounted prices. By exploiting the diversity in this market, Cloud users can utilize spot instances as long as possible to reduce their cost further. Recently, Amazon introduced the Spot Fleet API [12], which allows users to bid for a pool of resources at once. The provision of resources is automatically managed by Amazon using a combination of spot instances. However, it still lacks the fault-tolerant capability to avoid availability and performance impact caused by the sudden termination of spot instances, and thus, is not suitable to provision web applications.

To fill in this gap, we aim to build a solution to cater this need. We proposed a reliable auto-scaling system for web applications using heterogeneous spot instances along with on-demand instances. Our approach not only significantly reduces the financial cost of using Cloud resources but also ensures high availability and low response time, even when some types of spot VMs are early terminated by Cloud provider simultaneously or consecutively within a short period.

The contributions of this chapter are:

- a fault-tolerant model for web applications provisioned by spot instances;
- cost-efficient auto-scaling policies that comply with the defined fault-tolerant semantics using heterogeneous spot instances;
- event-driven prototype implementations of the proposed auto-scaling system on CloudSim [30] and Amazon EC2 platform;
- performance evaluations through both repeatable simulation studies based on historical data and real experiments on Amazon EC2.

The remainder of the chapter is organized as follows. We first model our problem in Section 5.2. In Section 5.3, we propose the base auto-scaling policies using heterogeneous spot instances under hourly billed context. Section 5.4 explains the optimizations we proposed on the initial policies. Section 5.5 briefly introduces our prototype imple-

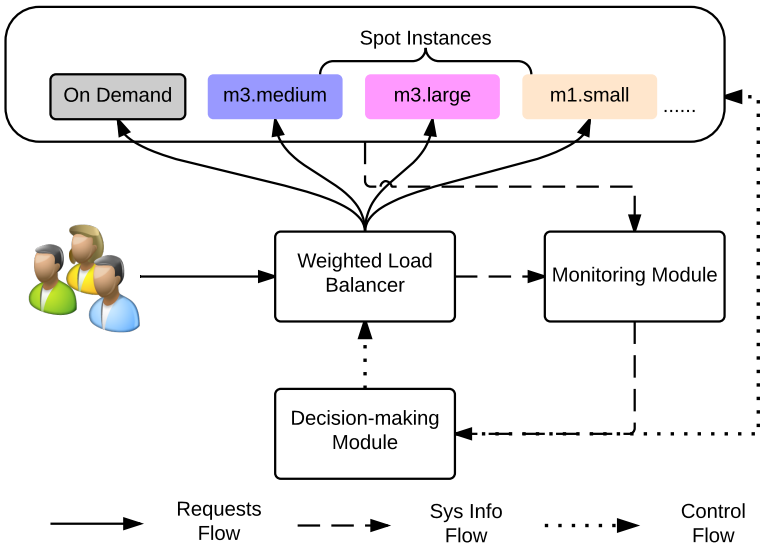


Figure 5.2: Proposed Auto-scaling system architecture

mentations. We present and analyze the results of the performance evaluations in Section 5.6 and discuss the related works in Section 5.7. Finally, we summarize the chapter.

5.2 System Model

For reader's convenience, the symbols used in this chapter are listed in Table 5.1.

5.2.1 Auto-scaling System Architecture

As illustrated in Figure 5.2, our auto-scaling system provisions a single-tier (usually the application server tier) of an application using a mixture of on-demand instances and spot instances. The provisioned on-demand instances are homogeneous instances that are most cost-efficient to run the application, while spot instances are heterogeneous.

Like other auto-scalers, our system is composed of the *monitoring* module, the *decision-making* module, and the *load balancer*. The monitoring module consists of multiple independent monitors that are responsible for fetching newest corresponding system information such as resource utilizations, request rates, spot market prices, and VMs' statuses. The decision-making module then makes scaling decisions according to the obtained information based on the predefined strategies and policies when necessary. Since in our

Table 5.1: List of Symbols

Symbol	Meaning
T	The set of spot types
M_{min}	The minimum allowed resource margin of an instance
M_{def}	The default resource margin of an instance
Q	The quota for each spot group
R	The required resource capacity for the current load
F_{max}	The maximum allowed fault-tolerant level
f	The specified fault-tolerant level
O	The minimum percentage of on-demand resources in the provision
S	The maximum number of selected spot groups in the provision
r_o	The resource capacity provisioned by on-demand instances
s	The number of chosen spot groups
vm	The VM type
vm_o	The on-demand VM type
c_{vm}	The hourly on-demand cost of the vm type instance
$num(c, vm)$	The function returns the number of vm type instances required to satisfy resource capacity c
C_o	The hourly cost of provision in on-demand mode
tb_{vm}	The truthful bidding price of vm spot group
m	The dynamic resource margin of an instance

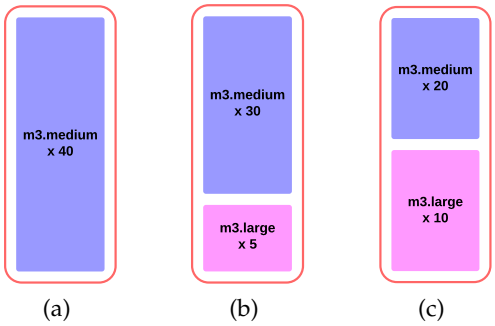


Figure 5.3: Naive provisioning using spot instances[1]

proposed approach, the provisioned virtual cluster is heterogeneous, the load balancer should be able to distribute requests according to the capability of each attached VM. The algorithm we use in this case is *weighted round robin*.

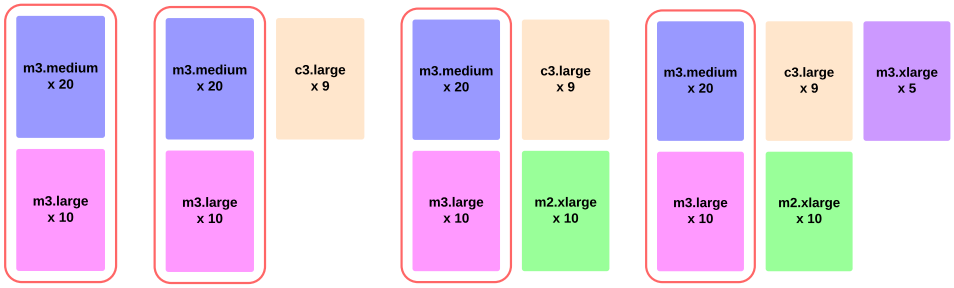
The application managed by the proposed auto-scaler should be stateless. This restriction does not reduce the applicability of our approach as modern Cloud applications should be developed in a stateless framework to realize high scalability and availability [210]. Besides, stateful applications can be quickly transformed into stateless services using various means, e.g., storing the session data in a separate Memcached cluster.

5.2.2 Fault-Tolerant Mechanism

Suppose there are sufficient temporal gaps between price variation events of various types of spot VMs, increasing spot heterogeneity in provision can improve robustness. As illustrated in Figure 5.3a, the application is fully provisioned using 40 *m3.medium* spot VMs only, which may lead it to lose 100% of its capacity when *m3.medium*'s market price go beyond the bidding price. By respectively provisioning 75% and 25% of the total required capacity using 30 *m3.medium* and 5[2] *m3.large* spot VMs in Figure 5.3b, it will lose at most 75% of its processing capacity when the price of either chosen type rises above the bidding price. Furthermore, if it is provisioned with equal capacity using the two types

[1]The red rectangles in Figure 5.3, 5.4, 5.5, and 5.6 stand for the minimum amount of capacity required to process the current workload. Its value is dynamic and proportional to the changing workload so as the amount of redundancy for fault-tolerance.

[2]According to Amazon's specification, the capacity of 1 m3.large instance is equal to the capacity of 2 m3.medium instances.



(a) f-0 (b) f-1 (c) f-2 (d) f-3

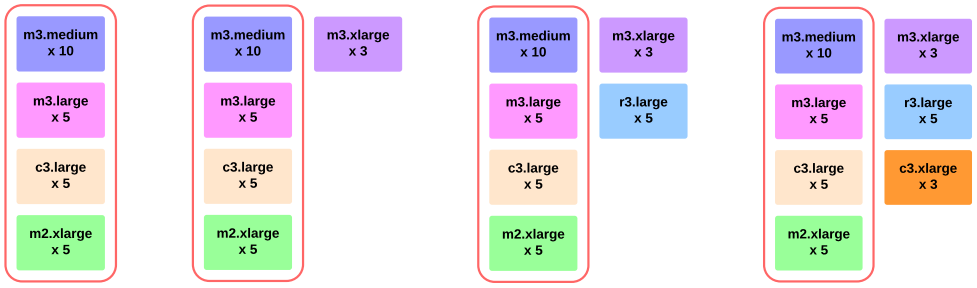
Figure 5.4: Provisioning for different fault-tolerant levels

of spot VMs, like in Figure 5.3c, termination of the either type of VMs will only cause it to lose 50% of its capacity.

It is still unsatisfactory as we demand application performance to be intact even when early termination happens. The solution is to further over-provision the same amount of capacity using another spot type, as the example illustrated in Figure 5.4b, it can be 50% of the required capacity provisioned using nine *c3.large* instances. In this way, the application is now able to tolerate the terminations of any utilized type of VMs and remain fully provisioned. After detection of the terminations, our auto-scaler can either provision the application using another kind of spot VMs or switch to on-demand instances. Application performance is unlikely to be affected if no other termination happens before the scaling operation that repairs the provision fully completes.

However, it takes considerable time to acquire and boot a VM (around 2 minutes for on-demand instances and 12 minutes for spot instances [138]). Hence, there is a substantial possibility that another type of spot VMs could be terminated within this time window. To counter such situation, it requires to over-provision the application using more spot types. We define the ***fault-tolerant level*** of our auto-scaler as the maximum number of spot types that can be early terminated without affecting application performance before its provision can be fully recovered. Figure 5.4 respectively shows the provision examples that comply with fault-tolerant level zero, one, two, and three in our definition with each spot type provisioning 50% of the required capacity.

Note that setting fault-tolerant level to zero is usually not recommended. Though using multiple types of spot instances confines amount of resource loss when failures



(a) f-0 (b) f-1 (c) f-2 (d) f-3

Figure 5.5: Provisioning for different fault-tolerant levels using 2 more spot types

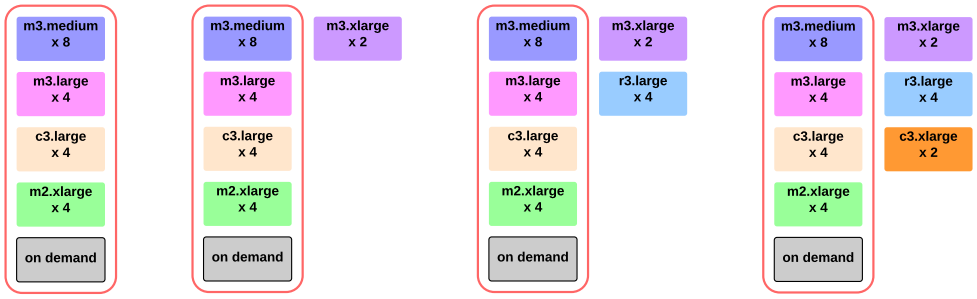
happen, with no over-provision to compensate resource loss, it may frequently cause performance degradation as failure probability becomes higher when more types of spot instances are involved.

5.2.3 Reliability and Cost Efficiency

Although the provisions shown in Figure 5.4b, 5.4c, and 5.4d successfully increase reliability of the application, they are not cost-efficient. The three provisions respectively over-provision 50%, 100%, and 150% of resources required by the application, which greatly diminishes the cost saving of using spot instances.

One possible improvement is to provision the application using more number of spot types. The illustrative provisions in Figure 5.5 employ two more spot types than that are used in Figure 5.4 to reach the corresponding fault-tolerant levels. As a result, total over-provisioned capacities for the three cases are reduced to 25%, 50%, and 75%. Though the provisions now might become more volatile with more types of spot VMs involved, the increased risk is manageable by the fault-tolerant mechanism with over-provision.

Another choice is to provision the application with a mixture of on-demand instances and spot instances to reduce over-provision . Like the demonstrations shown in Figure 5.6, there are now only 20%, 40%, and 60% over-provisioned capacities if on-demand instances provision 20% of the required resource capacity. Moreover, using on-demand resources also further confines amount of capabilities that could lose unexpectedly, thus, improving robustness. On the other hand, this method incurs a more financial cost.



(a) f-0 (b) f-1 (c) f-2 (d) f-3

Figure 5.6: Provisioning for different fault-tolerant levels using mixture of on-demand and spot instances

We define total capacity that is provisioned by the same type of spot VMs as a ***Spot Group***. In addition to that, we give definition to ***Quota*** (Q), which is the capacity each spot group needs to provision given the capacity provisioned by on-demand resources (r_o) and the fault-tolerant level (f). It is calculated as:

$$Q = \frac{R - r_o}{s - f} \tag{5.1}$$

where R represents the required capacity for the current load and s denotes the number of chosen spot types. The minimum amount of capacity that is required to over-provision then can be calculated as $Q * f$.

We call a provision is ***safe*** if the provisioned capacity of each spot group is larger than Q. Hence, the problem of scaling web applications using heterogeneous spot VMs is transformed to dynamically selecting spot VM types and provisioning corresponding spot and on-demand VMs to keep the provision in the safe state with minimum cost when the application workload increases, and timely deprovisioning various types of VMs when they are no longer needed.

5.3 Scaling Policies

Based on the previous fault-tolerant model, we propose cost-efficient auto-scaling policies that comply with the defined fault-tolerant semantics for hourly billed Cloud market

like Amazon EC2.

5.3.1 Capacity Estimation and Load Balancing

Our auto-scaler is aware of multiple resource dimensions (such as CPU, Memory, Network, and Disk I/O). It needs the profile of the target application regarding its average resource consumption for all the considered dimensions. Currently, the profiling needs to be performed offline, but our approach is open to integrate dynamic online profiling into it.

With the profile, the auto-scaler can estimate the processing capability of each spot type under the context of the scaling application. Based on that, it can quickly determine how to distribute incoming requests to the heterogeneous VMs to balance their loads. Besides, the estimated capabilities are used in the calculation of scaling plans as well.

5.3.2 Spot Mode and On-Demand Mode

Our auto-scaler runs interchangeably in ***Spot Mode*** and ***On-Demand Mode***. Spot Mode provisions application in the way explained in Section 5.2.3. In Spot Mode, the user needs to specify the minimum percentage of required resources provisioned by on-demand instances, symbolized as O. He can also set a limit on the number of selected spot groups in the provision, denoted as S. To define these parameters, users can utilize the simulation tool implemented by us (described in Section 5.5) to find the optimal configurations according to the recent spot market history without running real tests on the Cloud. Furthermore, these parameters can be dynamically adjusted using machine learning techniques. In On-Demand Mode, the application is fully provisioned by on-demand instances without over-provision. Switches between modes are dynamically triggered by the scaling policies detailed in the below sections.

5.3.3 Truthful Bidding Prices

Bidding truthfully means that participants in an auction always bids the maximum price they are willing to pay. Truthful bidding price for each VM type in our policies is cal-

Algorithm 3: Find new provision when the system needs to scale up

Input: R : the current workload
Input: n_c : the number of on-demand VMs in current provision
Input: vm_o : the on-demand vm type
Input: O : the minimum percentage of on-demand resources
Output: *target_provision*

1 $min\_vm_o \leftarrow \mathbf{max}(n_c, num(R * O, vm_o))$;
2 $max\_vm_o \leftarrow num(R, vm_o)$;
3 *candidate_set* $\leftarrow$ call Algorithm 4 for each integer n in $[min\_vm_o, max\_vm_o]$;
4 **return** on-demand provision if *candidate_set* is empty
5 otherwise the provision with minimum cost in *candidate_set*;

culated dynamically according to real-time workload and provision. Before computing them, we first calculate the hourly baseline cost if the application is provisioned in On-Demand Mode, which can be represented as:

$$C_o = num(R, vm_o) * c_{vm_o} \tag{5.2}$$

where the function $num(R, vm_o)$ returns the minimum number of instances of on-demand VM type required to process the current workload. c_{vm_o} is the on-demand hourly price of the on-demand instance type. Then truthful bidding price of spot type vm is determined as follow:

$$tb_{vm} = \frac{C_o - num(r_o, vm_o) * c_{vm_o}}{s * num(Q, vm)} \tag{5.3}$$

where $num(r_o, vm_o)$ and $num(Q, vm)$ are interpreted similarly to $num(R, vm_o)$ in Equation (5.2).

It ensures that even in the worst situation that all chosen spot types' market prices are equal to their corresponding truthful bidding prices, the total hourly cost of the provision will not exceed that in On-Demand Mode.

Algorithm 4: Find provision given the number of on-demand instances

Input: n : the number of on-demand VMs
Input: g_c : the set of spot groups in current provision
Input: vm_o : the on-demand vm type
Input: f : the fault-tolerant level
Input: **T** : the set of spot types
Input: S : the maximum number of chosen spot groups
Output: *new_provision*

$min\_groups \leftarrow \mathbf{max}(|g_c|, f+1)$;
$max\_groups \leftarrow \mathbf{min}(|\mathbf{T}|, S)$;
if $max\_groups < min\_groups$ **then**
    provision not found;
end
else
    for s ***from*** min_groups ***to*** max_groups **do**
        $p \leftarrow p \cup (vm_o, n)$;
        compute Q using Equation (5.1);
        compute tb_{vm} for each vm in **T**;
        $p \leftarrow p \cup g_c$;
        $groups \leftarrow$ each $group$ not in g_o and whose tb_{vm} is higher than market price;
        $k \leftarrow s - |g_c|$;
        if $|groups| \geq k$ **then**
            $p \leftarrow p \cup$ top k cheapest groups in $groups$;
            $provisions \leftarrow provisions \cup p$;
        end
    end
end
return the cheapest provision in *provisions*;

5.3.4 Scaling Up Policy

Scaling up policy is called when some instances are early terminated, or the current provision cannot satisfy resource requirement of the application. By resource requirement, in Spot Mode, it means the provision should be ***safe*** under the current workload, which is defined in Section 5.2.3. While in On-Demand Mode, it requires the resource capacity of the provision to exceed the resource needs of the current workload.

Algorithm 3 is used to find the ideal new provision when the system needs to scale up.

The algorithm only provisions VMs incrementally to avoid frequent drastic changes. As shown by line 1 in Algorithm 3, it limits the number of provisioned on-demand instances to be at least its current number. For each valid number of on-demand instances, it calls Algorithm 4 to find the corresponding best provision among provisions with various combinations of spot groups. Similarly, in Algorithm 4 (line 11), it retains the spot groups chosen by the current provision and only incrementally adds new groups according to their cost-efficiency (line 15). If there is no valid provision found, the auto-scaler switches to on-demand mode.

After the target provision is found, the auto-scaler compares it with the current provision and then contacts the Cloud provider through its API to provision the corresponding types of VMs that are in short.

In the worst case, the time complexity of the scaling up policy is $O(N * S * |\mathbf{T}|))$ where N is the number of on-demand instances required to provision the current workload in on-demand mode, S denotes the maximum number of chosen spot groups, and $|\mathbf{T}|$ is the number of spot types considered. Since the parameters are all small integers, the computation overhead of the algorithm is acceptable in an online decision-making scenario.

5.3.5 Scaling Down Policy

Since each instance is billed hourly, it is unwise to shut down one instance before its current billing hour matures. We, therefore, put the decision of whether each instance should be terminated or not at the end of their billing hours. The particular decision algorithms are different for on-demand instances and spot instances.

Policy for On-Demand Instances

When one on-demand instance is at the end of its billing hour, we not only need to decide whether the instance should be shut down but also have to make changes to the spot groups if necessary. The summarized policy is abstracted in Algorithm 5. The algorithm first checks whether enough on-demand instances are provisioned to satisfy the on-demand capacity limit (line 1 and line 2). If there are sufficient on-demand instances,

Algorithm 5: Find target provision when the billing hour of one on-demand instance is about to end

Input: R : the current workload
Input: n_c : the number of on-demand instances in current provision
Input: vm_o : the on-demand vm type
Input: O : the minimum percentage of on-demand resources
Output: *target_provision*

```
1 if n_c ≤ num(R * O, vm_o) then
2 |   provision not found;
3 end
4 else
5 |   p_1 ← call Algorithm 4 with n_c;
6 |   p_2 ← call Algorithm 4 with n_c − 1;
7 |   return on-demand provision if neither p_1 nor p_2 is found otherwise either
  |     provision that is cheaper;
8 end
```

it endeavours to find the most cost-efficient provisions with and without the on-demand instance by calling Algorithm 4 (line 5 and line 6). Suppose the current provision is in On-Demand Mode and no provision is found without the on-demand instance, the provision will remain in On-Demand Mode. Otherwise, if a new provision is found without the current instance, the policy switches the provision to Spot Mode. In the case that the current provision is already in Spot Mode, it picks whichever provision that incurs a lower hourly cost.

Policy for Spot Instances

When dealing with a spot instance whose billing period is ending, in the base policy, we simply shut down the instance when the corresponding spot quota Q can be satisfied without it. The base policy will evolve with the introduced optimizations in Section 5.4

5.3.6 Spot Groups Removal Policy

Note that in both scaling up and down policies; we forbid removing selected spot groups from the provision. Instead, we evict a chosen spot group when the provider terminates

any spot instances of such type. Since bidding price of each instance is calculated dynamically, instances within the same spot group may be bid at different prices, which could cause some instances to remain alive even after the corresponding spot groups are removed from the provision. We call the instances that are running but do not belong to any group ***orphans***. Though orphan instances are still in production, they are not considered a part of the provision according to the fault-tolerant semantics when making scaling decisions. In the base policies, although they will not be shut down until their billing hour ends, new instances still need to be launched to comply with the fault-tolerant semantics, which causes resource waste. The introduced optimizations address this drawback in the following section.

5.4 Optimizations

We have made several optimizations on the above base policies to improve cost-efficiency and reliability of our auto-scaler further.

5.4.1 Bidding Strategy

In our scaling policies, spot groups are bid at truthful bidding prices calculated by Equation (5.3) due to cost-efficiency concern. While focusing on robustness, the system can employ a different strategy to bid higher to retain spot instances as long as possible.

Actual Bidding Strategies

There are two bidding strategies, namely truthful bidding strategy and on-demand price bidding strategy embedded in the system.

- **Truthful Bidding Strategy:** the system always bids the truthful bidding price calculated by Equation (5.3) when new spot instances are launched. Since partial billing hours ended by Cloud provider are free of charge, Cloud users can save money by letting Cloud provider terminate their spot instances once their market prices ex-

ceed the corresponding truthful bidding prices. On the other hand, it leads to more early terminations.

- **On-Demand Price Bidding Strategy:** the system always bids the on-demand price of the corresponding spot type whenever trying to obtain new spot instances. This strategy will cost Cloud users more money but provides a higher level of protection against early terminations.

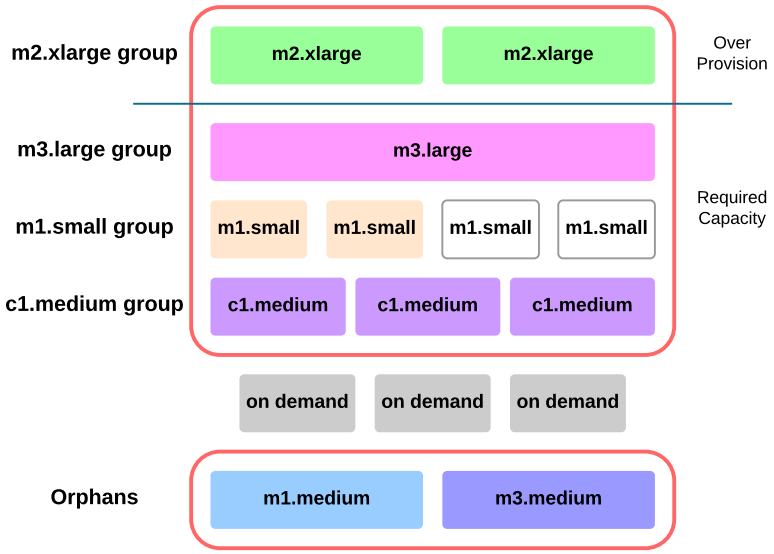
Revised Spot Groups Removal Policy

In the base policies, less cost-efficient spot groups could remain in the provision for a long time unless provider terminates some of their instances. When the actual bids are higher than the truthful bidding prices, the situation could become worse. Instead of just relying on the provider terminating uneconomical spot groups, the revised policy actively inspects whether market prices of some spot groups have exceeded their corresponding truthful bidding prices and remove them from the provision. In the meantime, for spot groups whose market prices are still below their truthful bidding prices, it looks for the chance to replace them by more economical spot groups that have not been selected. Such operations should be conducted in a long interval, such as every 30 minutes in our implementation, to minimize disturbance to provision. Members of removed or replaced spot groups become orphans.

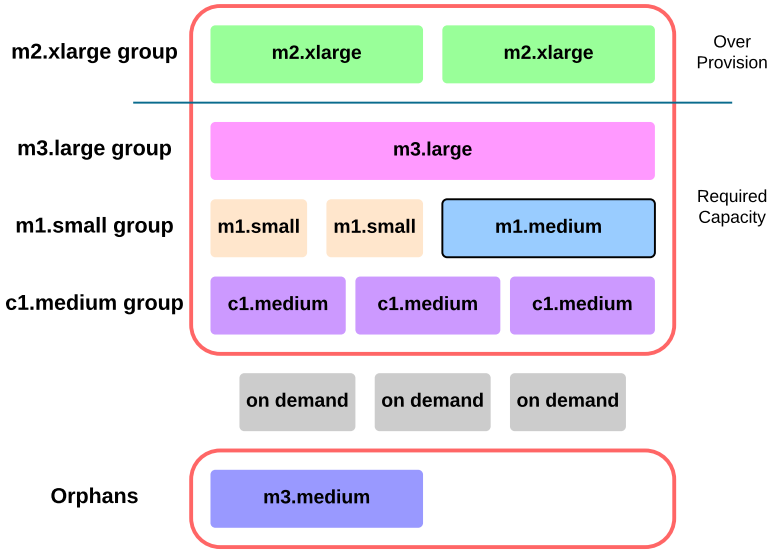
5.4.2 Utilizing Orphans

After removing or replacing some spot groups, if the system simply lets members of these spot groups become orphans and immediately start instances of newly chosen spot groups, the stability of provision will be affected. Furthermore, as orphans are not considered as valid capacity in the base policies, during the transition period, it has to provision more resources than necessary, which results in monetary waste.

To alleviate this problem, we aim to utilize as many orphans in the provision as possible to deter the time to provision new VMs. As a result, resource waste can be reduced, and cost-efficiency is improved.



(a) launching 2 new m1.small instances for m1.small spot group



(b) using one m1.medium orphan to temporarily substitute 2 m1.small instances

Figure 5.7: Provisioning with orphans under fault-tolerant level one

We modify the proposed fault-tolerant model to allow a spot group temporarily accept instances that are heterogeneous to the spot group type under certain conditions. Figure 5.7 illustrates such provision. In Figure 5.7a, the *m1.small* group does not have sufficient instances to satisfy its quota. Instead of launching 2 new *m1.small* spot instances, the policy now temporarily move the available orphan, one *m1.medium* instance, to the *m1.small* group to compensate the deficiency of its quota. Even though *m1.small* group becomes heterogeneous in this case, it does not violate the fault-tolerant semantics as losing any type of spot instances will not influence the application performance. However, in some situations, heterogeneity in spot groups could cause violation of the fault-tolerant semantics, for example, there might be case that three *m1.medium* orphans are spread across three spot groups, and the total capacity of the three instances exceeds the spot quota. Then losing the three *m1.medium* instances will violate the fault-tolerant semantics. Fortunately, such cases are very rare as orphans are usually small in numbers and are expected to be shut down soon.

With this relaxation of the fault-tolerant model, the previous scaling up and scaling down policies need to be revised to utilize the capacities of orphans efficiently.

Revised Scaling Up Policy

The new scaling up policy uses the same algorithm (Algorithm 3) to find the target provision. However, instead of launching instances to reach the objective provision, the new policy calculates whether it can utilize existing orphans to meet the quota requirements in the target provision.

The new policy first checks whether the destination provision chooses new spot groups. If there are orphans whose types are the same to any newly selected groups, lying either within the orphan queue or other spot groups, they are immediately moved to the corresponding new spot groups. After that, the policies endeavor to insert non-utilized orphans from the orphan queue into spot groups that have not met their quota requirements. If all the orphans have been utilized and some groups still cannot satisfy their quota, new spot instances of the corresponding types are launched.

Revised Scaling Down Policy

Regarding policy for on-demand instances that are close to their billing hour, the new policy utilizes the same mechanism in the revised scaling up policy to provision any changes between the current provision and the target one.

For the spot scaling down policy, if the spot instance is in the orphan queue, it is immediately shut down. Suppose it is within the spot group of the same type, it is shut down when the spot quota can be satisfied without it. In the case that the instance is an orphan within another spot group, the new policy shuts down the instance and in the meantime starts a certain number of spot instances of the spot group type to compensate the capacity loss.

5.4.3 Reducing Resource Margin

For applications running on a traditional auto-scaling platform, administrator usually leaves a margin for each instance to handle short-term workload surge to buy time for booting up new instances. This margin empirically ranges from 20 to 25% of the instance's capacity.

With over-provision already in place, this margin can be reduced under Spot Mode provision. We devise a mechanism that dynamically changes the margin according to the current fault-tolerant level. Since higher fault-tolerant level leads to more over-provision, we can be more aggressive in reducing the margin of each instance. In detail, the dynamic margin is determined by the formula:

$$m = \frac{M_{def} - M_{min}}{F_{max}} * f + M_{min} \tag{5.4}$$

where M_{min} means the minimum allowed margin, e.g., 10%, M_{def} is the default margin used without dynamic margin reduction, e.g., 25%, and F_{max} is the maximum allowed fault-tolerant level.

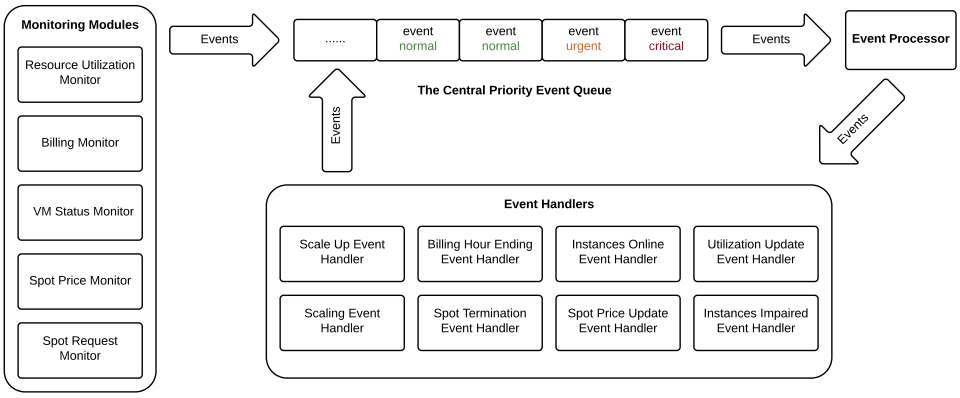


Figure 5.8: Components of the Implemented Auto-scaling System

5.5 Implementation

We implemented a prototype of the proposed auto-scaling system on Amazon EC2 platform using Java, the components of which are illustrated in Figure 5.8. It employs an event-driven architecture with the monitoring modules continuously generating events according to newly obtained information, and the central processor consuming events one by one. Monitoring modules produce and insert corresponding events with various critical levels into the central priority event queue. They include the *resource utilization monitors* that watch all dimensions of resource consumption of running instances, the *billing monitor* that gazes billing hour of each requested VM, the *VM status monitor* that reminds the system when some instances are online or offline, the *spot price monitor* that records newest spot market prices for each considered spot type, and the *spot request monitor* that watches for early spot termination. On the other side, the central event processor fetches events from the event queue and assigns them to the corresponding event handlers that realize the proposed policies to make scaling decisions or perform scaling actions.

The prototype implementation provides a general interface for users to plug different load balancer solutions into the auto-scaler. In our case, we use *HAProxy* with weighted round robin algorithm. It also offers the interface to allow users to automatically customize configurations of VMs according to their available resources after they have been booted.

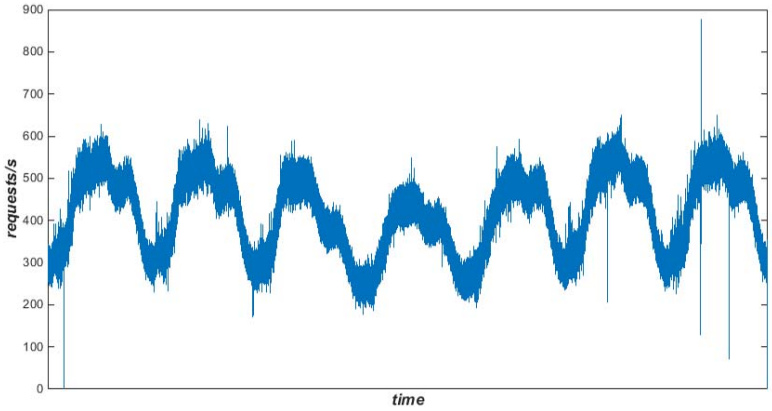


Figure 5.9: The English Wikipedia workload from Sep 19th 2009 to Sep 26th 2009

For quick concept validation and repeatable evaluation of the proposed auto-scaling policies, we created a simulation version of the system. The same code base is transplanted onto CloudSim [30] toolkit which provides the underlying simulated Cloud environment. The simulation tool can provide quick and economic validation of the proposed policies using historical data of the application and the spot market as input if bids from a single user impose negligible influence on market prices.

5.6 Performance Evaluation

5.6.1 Simulation Experiments

As stated in Section 5.5, to allow repeatable evaluation, we developed a simulation version of the system that allows us to compare the performances of different configurations and policies using traces from real applications and spot markets.

Simulation Settings

We use one week trace of 10% English Wikipedia requests from Sep 19th 2007 to Sep 26th 2007 as the workload [201, 204], which is depicted in Figure 5.9. Note that our approach is general purpose and can be applied to any workload, as the proposed system does not make assumptions on the workload and is fully reactive. We adopt the Wikipedia

workload in experiments because it reveals significant variations that can trigger frequent scaling operations to let us observe the behavior of our system. We believe one week trace is enough for the purpose of our experiments, as it gives the system ample opportunities to exercise the scaling policies. Besides, as reported by Eldin et al. [60], the Wikipedia workload revealed strong weekly pattern with only gradual changes in amplitude, level, and shapes.

We consider 13 spot types in Amazon EC2. Their spot prices are simulated according to one week Amazon's spot prices history from March 2nd, 2015 18:00:00 GMT in the relatively busy ***us-east*** region. The involving spot types and their corresponding history market prices are illustrated in Figure 5.1.

We set request timeout at 30 seconds. In addition, we respectively set minimum allowed resource margin (M_{min}) and default resource margin (M_{def}) at 10% and 25%. We found out that the *c3.large* is the most cost-efficient type to run Wikipedia application according to a small-scale resource profiling test of the Wikibench application [203] on Amazon EC2 and the resource specifications of each instance type released by Amazon. It is selected to provision all the on-demand resources in the experiments. All simulation experiments start with 5 *c3.large* on-demand instances. The length of simulated requests is generated following a pseudo-Gaussian distribution[3] with a mean of 0.07 ECU[4] and standard deviation of 0.005 ECU so that different tests using the same random seed are receiving the same workload. VM startup, shut down, and spot requesting delays are generated in the same way using pseudo-Gaussian distribution. The mean values of the above three distributions are respectively 100, 100, 550 seconds, and the standard deviations are set at 20, 20, 50 seconds. The test results are deterministic and repeatable on the same machine.

We tested our scaling policies with various fault-tolerant levels and different least amounts of on-demand resources, which are represented respectively as "$f - x$" and "y% on-demand" in the results. We also tested the policies using the two embedded bidding

[3]Since Wikipedia is serving mostly the same type of requests - page view, the time taken to process each request is also likely to fall in a particular interval. To coarsely model such behavior, we utilize Gaussian distribution. Other distributions with small head and tail can serve the same purpose as well.

[4]It means the request takes 70ms to finish if it is computed by the VM equipped with vCPU as powerful as 1 Elastic Computing Unit (ECU)

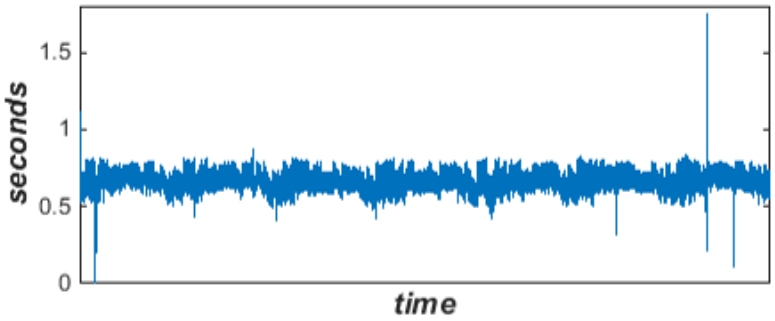


Figure 5.10: Response time for on-demand auto-scaling

strategies and static/dynamic resource margins.

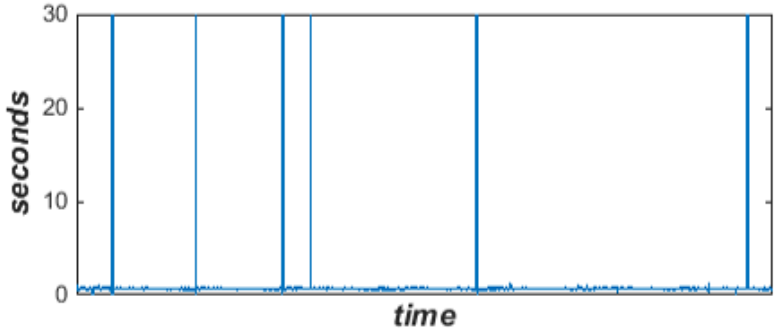
We recorded two metrics, real-time response time of requests (average response time per second reported) and total cost of instances, in all the experiments.

Benchmarks

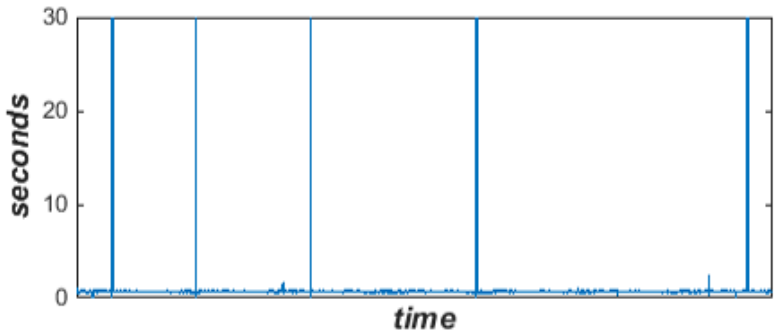
We compared our scaling policies with two benchmarks:

- **On-Demand Auto-scaling:** This benchmark only utilizes on-demand instances. It is implemented by restricting the auto-scaler always in On-Demand Mode.

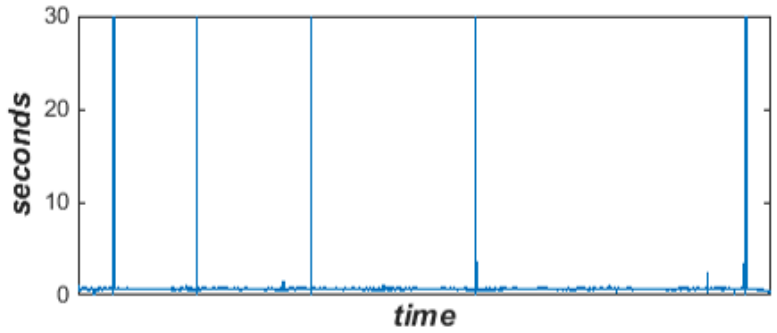
- **One Spot Type Auto-scaling:** The auto-scaling policies used in this benchmark, like the proposed policies, provision a mixture of on-demand resources and spot resources. The benchmark also has a limit on minimum amount of on-demand resources provisioned. However, for spot instances, it only provisions one spot group that is the most cost-efficient at the moment without over-provision. If the provisioned spot instances are terminated, a new spot group then is selected and provisioned. Suppose a more economic spot group is found, the old spot group is gradually replaced by the new one. It is implemented by setting the fault-tolerant level to zero and limiting at most one spot group can be provisioned.



(a) 0% on-demand resources

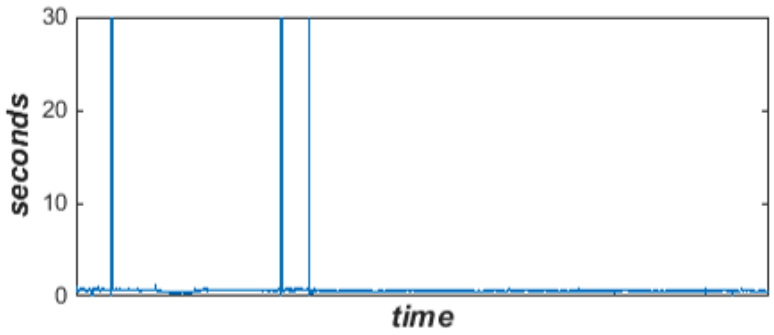


(b) 20% on-demand resources

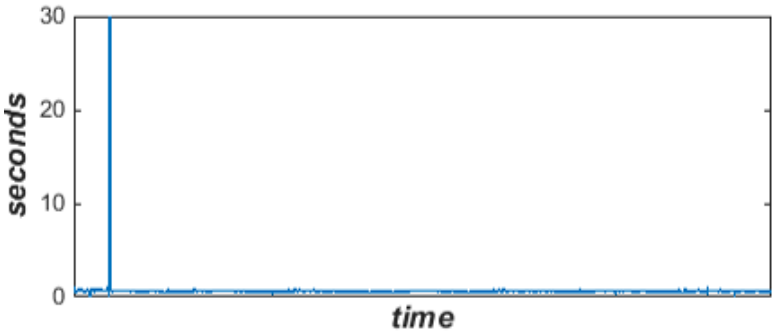


(c) 40% on-demand resources

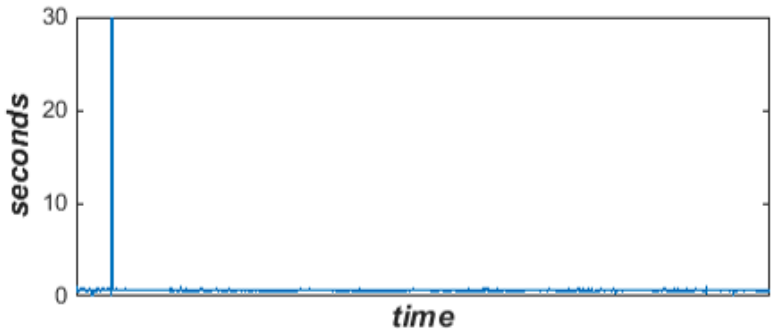
Figure 5.11: Response time of one spot type auto-scaling with various percentage of on-demand resources and truthful bidding strategy



(a) 0% on-demand resources

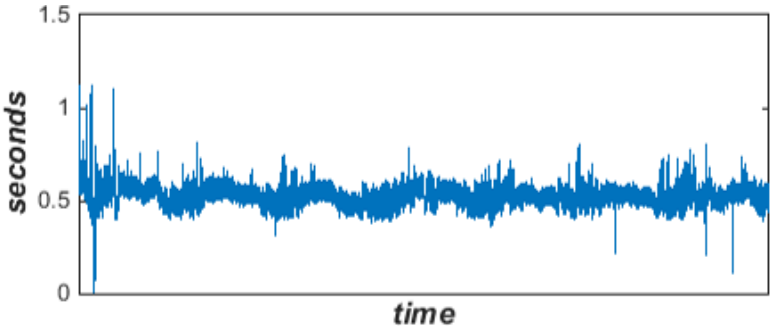


(b) 20% on-demand resources

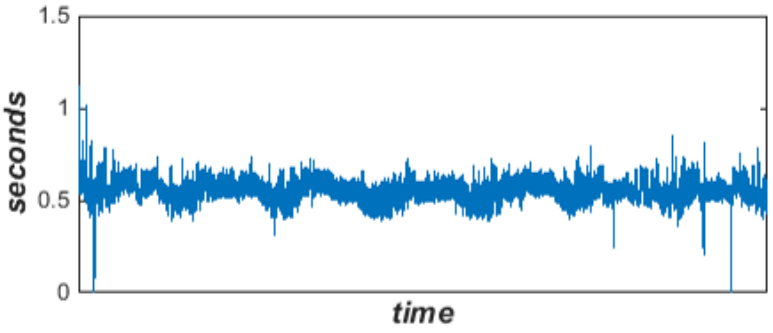


(c) 40% on-demand resources

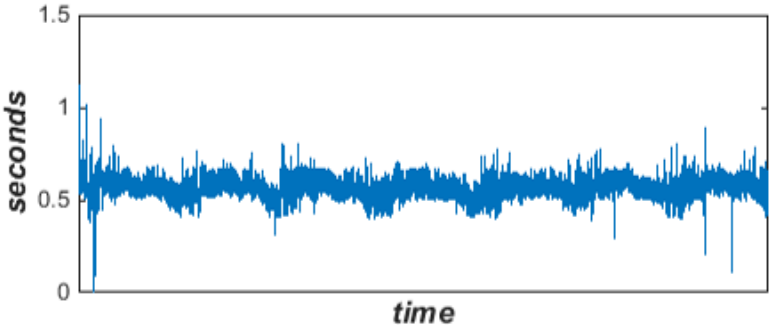
Figure 5.12: Response time of $f - 0$ with various percentage of on-demand resources, truthful bidding strategy, and dynamic resource margin



(a) 0% on-demand resources

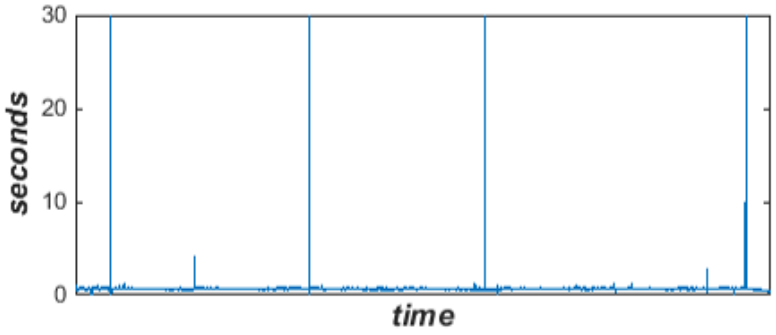


(b) 20% on-demand resources

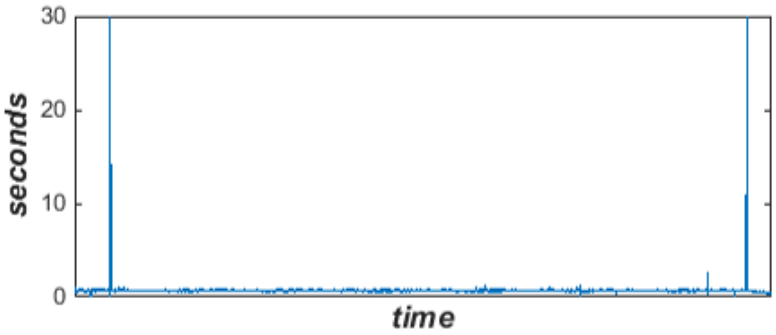


(c) 40% on-demand resources

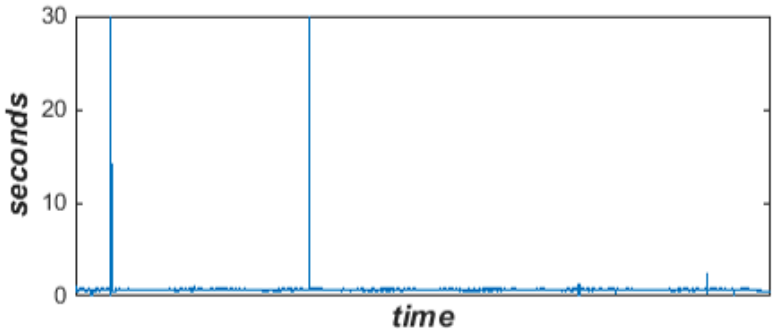
Figure 5.13: Response time of $f-1$ with various percentage of on-demand resources, truthful bidding strategy, and dynamic resource margin



(a) 0% on-demand resources



(b) 20% on-demand resources



(c) 40% on-demand resources

Figure 5.14: Response time of one spot type auto-scaling with various percentage of on-demand resources and on-demand bidding strategy

Response Time

Figures 5.10, 5.11, 5.12, and 5.13 respectively depict real-time average response time of requests using on-demand, one spot, and our approach with truthful bidding strategy and dynamic resource margin. From the results, the on-demand auto-scaling produced smooth response time all along the experimental duration except for a peak that was caused by the corresponding peak in the workload. All experiments employing one spot type auto-scaling experienced periods of request timeouts caused by the terminations of spot instances, and only increasing the amount of on-demand resources could not improve the situation. In contrast, our approach significantly reduced such unavailability of service even using $f-0$ with no over-provision of resources. By using $f-1$, we could eliminate the timeouts with the recorded spot market traces as input. We omit the results for tests using $f-2$ and $f-3$ as they reveal similar results as Figure 5.13.

To show the effect of different bidding strategies, we compare the response time results of one spot type auto-scaling using the two proposed bidding strategies as they reveal the most significant differences. As Figure 5.11 and Figure 5.14 present, it can be concluded that service availability can be much improved with higher bidding prices using one spot type auto-scaling. On the other hand, the remaining timeouts also indicate that increasing the bidding prices alone is not enough to guarantee high availability.

Cost

Table 5.2 lists the total costs produced by all the experiments. Comparing to the cost of on-demand auto-scaling, we managed to gain significant cost saving using all other configurations. Tests using one spot type auto-scaling with 0% on-demand resources enabled the most cost saving up to 80.87% regardless of its availability issue.

Results show the amount of on-demand resources has a significant influence on cost saving. It also can be noted that higher fault-tolerant level incurs extra cost. Though optimal configuration of the fault-tolerant level is always application-specific, according to our results, the configuration using $f-1$ with 0% on-demand resource is the best choice for the current market situation regarding both financial cost and service availability.

Table 5.2: Total Costs for Experiments with Various Configurations

USD$ / Policies	Total Cost			
on-demand	116.34			
	Truthful Bidding		**On-Demand Bidding**	
one spot with 0% **on-demand**	22.26		23.14	
one spot with 20% **on-demand**	46.50		47.33	
one spot with 40% **on-demand**	63.17		63.43	
$f-0$ **with** 0% **on-demand**	32.00		32.30	
$f-0$ **with** 20% **on-demand**	54.45		56.10	
$f-0$ **with** 40% **on-demand**	68.34		69.64	
	Static Resource Margin	**Dynamic Resource Margin**	**Static Resource Margin**	**Dynamic Resource Margin**
$f-1$ **with** 0% **on-demand**	41.57	39.32	43.17	41.66
$f-1$ **with** 20% **on-demand**	60.21	59.52	61.82	61.06
$f-1$ **with** 40% **on-demand**	72.09	72.55	72.96	73.08
$f-2$ **with** 0% **on-demand**	50.48	47.38	51.67	49.38
$f-2$ **with** 20% **on-demand**	67.72	65.52	68.71	66.09
$f-2$ **with** 40% **on-demand**	78.01	76.74	78.3	76.74
$f-3$ **with** 0% **on-demand**	67.87	62.61	68.79	61.50
$f-3$ **with** 20% **on-demand**	83.27	78.33	81.18	76.19
$f-3$ **with** 40% **on-demand**	89.86	85.57	88.09	84.46

The resulted cost differences caused by different bidding strategies are small. Therefore, it is better to bid higher to improve availability if user's bidding has a negligible impact on the market price.

As dynamic resource margin is only applicable when the application is over-provisioned, we present results for tests using dynamic resource margin when the fault-tolerant level is higher than zero. According to the results, dynamic resource margin can bring extra cost saving and the amount of cost saving increases when more over-provision is necessary (i.e., higher fault-tolerant level). Though the resulted cost saving is not significant, it is safely achieved without sacrificing availability and performance of the application.

5.6.2 Real Experiments

We conducted two real tests on Amazon EC2 respectively using on-demand auto-scaling policies and the proposed auto-scaling policies with a configuration of $f - 1$ and 0% on-demand. Other parameters are defined the same to the simulation tests.

We set up the experimental environment to run the Wikibench [203] benchmark tool. The major advantage of this tool compared to other tools such as TPC-W, RUBiS, and CloudStone is that it is stateless, which is characteristic of modern highly scalable Cloud services [210]. The tool is composed of three components:

- a client driver that mimics clients by continuously sending requests to the application server according to the workload trace;
- a stateless application server installed with the Mediawiki application;
- a MySQL database loaded with the English Wikipedia data by the date of Jan 3rd, 2008.

Our aim is to scale the application-tier. Thus, we inserted an HAProxy load balancer layer into the original architecture to let the client driver talk to a cluster of servers. The architecture of the testbed is illustrated in Figure 5.15. We picked the first three days of the Wikipedia workload [201, 204] (Figure 5.9) and scaled it down to half of its original

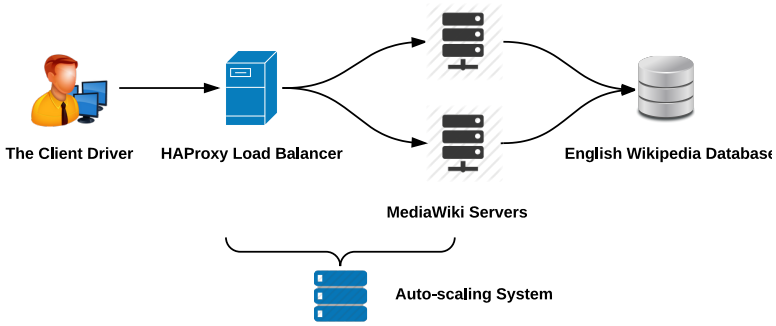


Figure 5.15: The Testbed Architecture

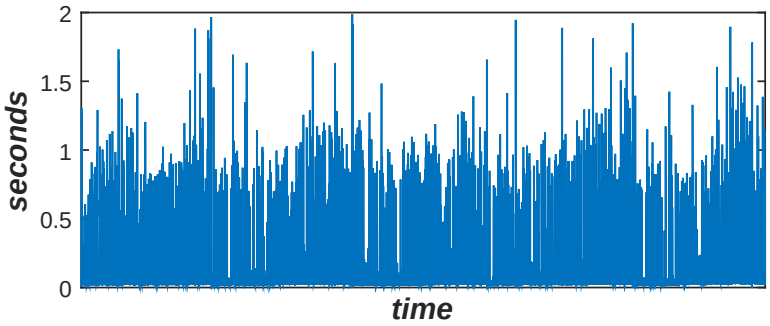


Figure 5.16: Response time for on-demand auto-scaling on Amazon

rate as the workload for testing because Amazon limits the number of instances each account can launch.

The testing environment resided in Amazon ***us-east-1d*** zone which is in a relatively busy region with higher degree and frequency of price fluctuations. Regarding each component, we launched one *c4.large* instance acting as the client driver, one *m3.medium* instance running the HAProxy load balancer, and one *c4.2xlarge*[5] instance serving the MySQL database requests. The auto-scaling system itself is running on a local desktop computer remotely in Melbourne. Before the tests, we profiled each component to make sure none of them become the bottleneck of the system.

The test using the proposed approach started at 3:30 am September 9, 2015, Wednesday, US Eastern time. The testing period spanned across three busy weekdays from Wednesday to Friday.

[5]The 4th generation instances were introduced between the time we performed the simulations and the real experiments. To be consistent, we only consider the 13 spot types listed in Figure 5.1 for both the simulations and the real experiments

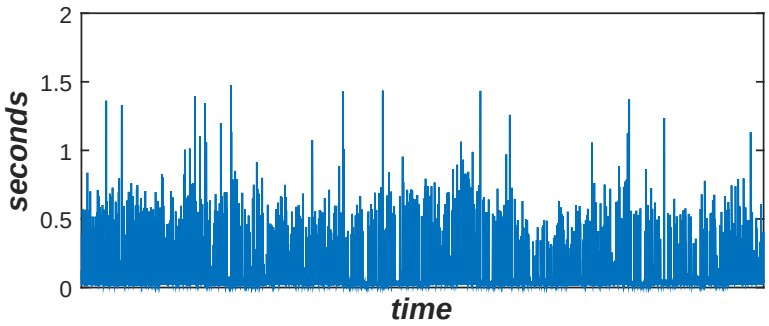


Figure 5.17: Response time for spot auto-scaling on Amazon

Table 5.3: Cost of the Experiments

Policies	**Cost(USD$)**
on-demand	19.01
$ft-1$ **and** 0% **on-demand**	5.69

Figure 5.16 and 5.17 presents real-time response time results of the two experiments. Both show peaks of high response time. By studying the recorded log, we confirmed shortage of resources did not cause them as resource utilizations of all the involving VMs were never beyond safe threshold during both tests and thus are not caused by our approach. We encountered three early terminations during the test of our approach. Thanks to the fault-tolerant mechanism and policies, we managed to avoid service interruption and performance degradation during those periods. Besides, because resources are tighter in on-demand auto-scaling, it performs worse in response time compared to the proposed approach.

Regarding cost, we calculated the total cost of application servers in both experiments. Table 5.3 presents the results. The proposed approach reaches 70.07% cost saving.

5.6.3 Discussion

Even with high fault-tolerant level, the proposed method cannot guarantee 100% availability, and no solution can ever manage to assure absolute service continuity due to the nature of spot market. What our system offers is the best effort to counter large-scale

surges of market prices of the selected spot types in a short time, which is highly unlikely under current market condition. In fact, we have not encountered any case that more than one spot group fail simultaneously during simulations, real experiments, and testing phases. However, the market condition could change. Hence, application providers should adjust the configuration of the auto-scaling system dynamically according to the real-time volatility of the spot market. Besides, the nature of the application also affects the decision. If the application is availability-critical, a higher fault-tolerant level is always desirable. Adversely, for some applications, such as analytical jobs, even one spot type auto-scaling is acceptable.

The presented results in Section 5.6 indicates the cost saving potential of an individual application considering a selected set of spot types under the recorded spot market prices and workload traces. Thanks to the dynamic truthful bidding price mechanism, even in competitive market condition, we can ensure that the cost reduction gained by our approach will not vanish but only diminish. To achieve more cost saving, the application provider can take into account a broader set of spot types, which is available in Amazon's offering.

To save cost and time for testing, application providers can tune the parameters of the auto-scaler by first utilizing simulation for fast validation and then test the system in the production environment.

There are also differences in price among the same spot types across different availability zones. It is trivial to extend the current fault-tolerant model to utilize spot groups from multiple availability zones. Currently, the auto-scaling system limits the selection of spot groups within the same availability zone due to charges for traffic across availability zones. If the application provider has already adopted a multi-availability-zone deployment, such extension can realize more cost saving.

The overhead of the auto-scaling system is negligible. As presented in Section 5.3, the time complexity of the scaling policies is not significant. The frequency that the scaling policies are called depending on the monitoring interval and the frequency of price changes, which are at least on the scale of seconds.

5.7 Related Work

5.7.1 Horizontally Auto-scaling Web Applications

Horizontally auto-scaling web applications have been extensively studied and applied [133]. In Chapter 2, we have provided a thorough taxonomy of the existing auto-scaling techniques.

Most industry auto-scaling systems are reactive-based. Among them, the most frequently used service is Amazon's Auto Scaling Service [11]. It requires user first to create an auto-scaling group, which specifies the type of VMs and image to use when launching new instances. Then the user should define his scaling policies as rules like "add two instances when CPU utilization is larger than 75%". RightScale offers Another popular service. Their service is based on a voting mechanism that lets each running instance decide whether it is necessary to grow or shrink the size of the cluster based on their condition [162].

Other than just using simple rules to make scaling decisions, researchers have developed scaling systems based on formal models. These models aim to answer the question that how many resources are required to serve a certain amount of incoming workload under QoS constraints. Such model can be obtained using profiling techniques as we did in this chapter. Other commonly adopted approaches include queuing models [68, 69, 82, 100, 166, 202] that either abstract the application as a set of parallel queues or a network of queues, and online learning approaches such as reinforcement learning [21, 27, 57].

Proactive auto-scaling is desirable because the time taken to start and configure newly started VMs creates a resource gap when workload suddenly surges to the level beyond the capability of the available resources. To satisfy strict SLA, sometimes it is necessary to provision enough resources before workload rises. As workloads of web applications usually reveal temporal patterns, accurate prediction of future workload is feasible using state-of-the-art time-series analysis and pattern recognition techniques. Many of them have been applied to auto-scaling of web applications [32, 58, 62, 87, 94, 100, 104, 164].

Most auto-scaling systems only utilize homogeneous resources, while some, includ-

ing our system, have explored the use of heterogeneous resources to provision web applications. Upendra et al. [200], and Srirama and Ostavar [182] adopt integer linear programming (ILP) to model the optimal heterogeneous resource configuration problem under SLA constraints. Fernandez et al. [63] utilizes tree paths to represent different combinations of heterogeneous resources and then searches the tree to find the most suitable scaling plan according to user's SLA requirements.

Distinct from the above works, our objective goes beyond using minimum resources to provision the application. Instead, we want to devise a fault-tolerant mechanism and auto-scaling policies that comply with the fault-tolerant semantics to reliably scale web applications on cheap spot instances. We believe the reviewed auto-scaling techniques are complementary to our approach. The proposed method can incorporate their resource estimation models, and workload prediction methods as well.

5.7.2 Application of Spot Instances

There have been many attempts to use spot instances to cut resource cost under various application context. Resource provisioning problems using spot instances have been studied for fault-tolerant applications [24, 35, 42, 43, 47, 134, 155, 183, 207, 220] such as high performance computing, data analytics, MapReduce, and scientific workflow.

For these applications, the fault-tolerant mechanism is often based on checkpointing, replication, and migration. Multiple innovative checkpointing mechanisms [96, 107, 167] have been developed to allow these applications to harness the power of spot instances. SpotOn [183] combines multiple fault-tolerant mechanisms to increase the cost-efficiency and performance of batch processing applications running on spot instances.

Regarding web applications, Zhao et al. [226] proposed a stochastic algorithm to plan future resource usage with a mixture of on-demand and spot instances. Besides the fact that they only use homogeneous resources, their object is also different to ours as they aim to plan the resource usage with the knowledge of the future while we provision resources dynamically. Mazzucco and Dumas [139] also explored the use of a mixture of homogeneous on-demand instances and spot instances to provision web applications. Instead of building a reliable auto-scaling system, their target is to maximize web ap-

plication provider's profit by using an admission control mechanism at the front end to dynamically adapt to sudden changes of available resources.

Sharma at al. proposed a derivative IaaS Cloud platform based on spot instances called SpotCheck [172, 176]. To transparently provide high availability on spot instances to end users, they incorporated technologies such as nested virtualization, live VM migration, and time-bounded VM migration with memory checkpointing, to dynamically move users' VMs when underlying spot instances are available or revoked. Because of its transparency to end users, it is ideal for Cloud brokers and large organizations with high resource demands, while our approach is lightweight and thus more suitable for small organizations who want to harness the power of spot instances by themselves. He et al. [86] from the same group evaluated the ability of the approach to reliably run web applications on spot instances. Though they do not provision redundant capacity as we do, they reported non-negligible overhead incurred by nested virtualization. Their proposed system [86, 172, 176] can preserve the memory state of the revoked spot VMs, which enables it to host stateful applications seamlessly. Though our approach requires applications to be stateless, this does not reduce its generality as highly scalable Cloud applications are expected to be stateless [210], and stateful applications can be transformed into stateless ones by storing session information in a memory cache cluster [210]. Their system relies on the termination warnings issued by existing providers [10] to be able to conduct migrations in time. Our approach is capable of operating in possible future spot markets that do not provide termination warnings.

Recently, Amazon EC2 introduced a new feature, called Spot Fleet API [12]. It allows a user to bid for a fixed amount of capacity, possibly constituted by instances of different spot types. It continuously and automatically provisions the capacity using the combination of instances. However, as its provision decision ignores reliability, it is not suitable to provision web applications.

5.8 Summary

In this chapter, we explored how to reliably and cost-efficiently auto-scale web applications using a mixture of on-demand and heterogeneous spot instances. We first proposed a fault-tolerant mechanism that can handle early spot terminations using heterogeneous spot instances and over-provision. We then devised novel cost-efficient auto-scaling policies that comply with the defined fault-tolerant semantics for hourly-billed Cloud market. We implemented a prototype of the proposed auto-scaling system on Amazon EC2 and a simulation version on CloudSim [30] for repeatable validation. We conducted both simulations and real experiments to demonstrate the efficacy of our approach by comparing the results with the benchmark approaches.

Though our proposed approach enhances the reliability of the application when facing terminated spot instances, short-term overload may still happen when many spot types are terminated simultaneously. In the next chapter, we present a technique that aims to reduce the performance impact caused by failures, like spot terminations, and flash crowds using geographical load balancing.

Chapter 6

Mitigating Impact of Overload on Web Applications through Geographical Load Balancing

Managed by an auto-scaler in the clouds, applications may still be overloaded by sudden flash crowds or resource failures as the auto-scaler takes time to make scaling decisions and provision resources. For applications deployed in multiple data centers, instead of sufficiently over-provisioning each data center to prepare for occasional overloads, it is more cost-efficient to over-provision each data center a small amount of capacity and to balance the extra load among them when resources in any data center are suddenly saturated. In this chapter, we present an approach that supplements and enhances current auto-scalers for applications deployed across multiple Clouds. It can timely detect overload situations and then autonomously handle them using the proposed geographical load balancing algorithm and admission control mechanism to minimize the resulted performance degradation. We developed a prototype and evaluated it on Amazon Web Services. The results show that our approach can maintain acceptable QoS while significantly increase the number of requests served during overloading periods.

6.1 Introduction

AUTO-SCALING is the mechanism that dynamically provisions resources to applications to cope the change of workloads or resource failures. However, it takes considerable time for auto-scalers to boot VMs and configure them for production. Mao and Humphrey [138] conducted an experimental study on the VM startup time of var-

This chapter is derived from: **Chenhao Qu**, Rodrigo N. Calheiros, and Rajkumar Buyya, "Mitigating Impact of Short-term Overload on Multi-Cloud Web Applications through Geographical Load Balancing", accepted by *Concurrency and Computation Practice and Experience*, 2017.

ious types of instances in Amazon AWS, Microsoft Azure, and Rackspace. They found that the booting time ranges from about 50s to more than 900s depending on the sizes, cost models, and operating systems. This delay in resource provisioning will result in performance degradation and even unavailability of service during this period.

In web applications, it is common to observe rapid surges in requests once in a while. This situation is called flash crowd, and it can occur any moment with little or no warning. The Cloud auto-scaler, in these cases, cannot timely provision enough resources to deal with these situations. Application providers now deploy web applications on cheap cost but unreliable Cloud resources, such as spot instances [12], which makes the applications more prone to resource failures. Therefore, solely relying on auto-scaling is not enough to ensure high performance all the times and a certain level of over-provisioning is necessary for production environments in preparation for these events.

Cloud providers have established their data centers all over the world, which enables their customers to deploy their applications in multiple geographically dispersed regions to better serve the worldwide population. As deployment of applications on multiple distributed computing Clouds is becoming increasingly popular, we argue that in this type of deployment, when failures happen, or a flash crowd arrives at a data center, it is better to utilize the unused capacities already provisioned in other data centers[1] to process as many exceeding requests as possible through geographical load balancing, instead of processing all the requests locally and degrading the performance of all the clients served by the regional data center, or rejecting the exceeding requests directly. This approach is viable as failures are unlikely to happen simultaneously in multiple data centers and flash crowds also seldom take place on a global scale at the same time due to culture and time differences.

A widely-adopted approach to implement geographical load balancing is through DNS resolution. However, it takes some time to populate the DNS settings across layered DNS servers, which makes it impossible to react to overload situations timely. Furthermore, it is also difficult to accurately control the load directed to each data center using

[1]Because load is balanced among all the servers provisioned in a data center, the unused capacities come from the spare processing capabilities in each server instead of from completely idle servers that are standing by.

this technique. Another popular way is to utilize a centralized load balancer to distribute load among data centers. Though it allows fine-grained control over traffic, it introduces extra latencies to all the requests, which reduces the benefit of deploying the application in multiple Clouds.

In this chapter, we present an approach that supplements and enhances state-of-the-art auto-scalers for applications deployed across multiple data centers. It aims to quickly detect and adapt to short-term overloads caused by resource failures and flash crowds in each data center through geographical load balancing and admission control before the auto-scaler finishes provisioning new resources. Different from previous geographical load balancing solutions, our approach relies on decentralized agents deployed in each data center to implement fast and accurate geographical load balancing. During the overload situations, the agent deployed in the overloaded data center temporarily forwards certain amounts of excessive requests to other data centers to keep the predefined SLA satisfied. Within our approach lies the proposed overload handling algorithm that optimally distributes excessive load among data centers that have available capacities causing a minimum overall increase of latencies. In this way, our approach only incurs little performance overhead to the forwarded requests and requests served by the receiving data centers during the short overloading periods, and thus preserves the benefit of deploying the application on multiple geographically distributed Clouds. We implemented a prototype and evaluated it on Amazon Web Services, who offers IaaS infrastructure in multiple geographically dispersed regions. Results show that our approach can timely detect short-term overload events and effectively improve the application performance during resource contention periods.

The contributions of the chapter are:

- a decentralized load balancing approach across multiple Clouds that detects and handles short-term overload events;
- an overload detection and handling algorithm;
- a queuing model for deciding load redirection among data centers with spare capacities to minimize its impact on the overall application performance; and

- a prototype implementation of the proposed approach evaluated in Amazon's infrastructure located in North Virginia, Ireland, and Tokyo.

The remainder of the chapter is organized as follows. In Section 6.2, we illustrate the use case scenarios of our approach. We then describe the deployment model and some assumptions in Section 6.3. We explain the proposed approach and its implementation in detail in Section 6.4. Finally, we compare our approach with related works and summarize the chapter in Section 6.7.

6.2 Use Case Scenarios

Our approach aims to mitigate performance degradation caused by short-term overloads that cannot be timely addressed by the Cloud auto-scalers. It is not designed to replace state-of-the-art auto-scalers; instead, it is complementary to them, and it should work cooperatively with them.

6.2.1 Resource Failures

Resource failures can happen at any time, and the failed resources will become inaccessible immediately, which leaves the auto-scaler no time to provision new resources without causing performance degradation during the provision time if the amount of resource loss is beyond the locally unused capacity. Such failures can frequently happen and in large scale if the application is deployed on unreliable resources, such as spot instances (a problem that is addressed in Chapter 5), which makes things worse. Spot instances are resources sold by Cloud providers through an auction-like mechanism. They are significantly cheaper than the corresponding on-demand instances, but they will be terminated by the provider when the market price exceeds the user bid, thus, causing failures. Besides infrastructure level, failures can also happen in software level, including the operating system, the application server, and the application itself. Our approach makes no assumption to the underlying failure types and can deal with them all.

If any failure is detected, the local agent in the overload handling framework and the

auto-scaler will intervene at the same time. During the failures, the agent temporarily forwards some requests to other data centers and enforces admission control if necessary to protect the application from crashing and maintain acceptable performance; meanwhile, the auto-scaler restarts the faulty resources or provisions new resources. When the provision process completes, and there are enough local resources, the agent then stops geographical load balancing and admission control.

6.2.2 Flash Crowds

Flash crowds might arrive anytime at any data center. They are difficult to be managed by auto-scalers alone due to their unpredictability and bursty nature. In the case of a flash crowd, widely used commercial auto-scalers, such as Amazon Auto-scaling Service [11], launch new VMs only after the application has experienced high load for a consecutive period set by the user, instead of provisioning new resources right after the detection of application overload. This mechanism is useful to reduce resource wastage and prevent scaling oscillations [133], as when the provision completes, the flash crowd might have already passed. Our approach can be an ideal partner of these commercial auto-scalers as it can help to reduce resource contentions not only during provision times but during waiting times as well.

6.3 Deployment Model and Application Requirements

6.3.1 Deployment Model

We assume the target application, including each of its composing services/components, is deployed in geographically dispersed data centers. Furthermore, the application instance in a data center should be able to communicate with instances located in other data centers through network for request forwarding purpose, which will be explained afterward.

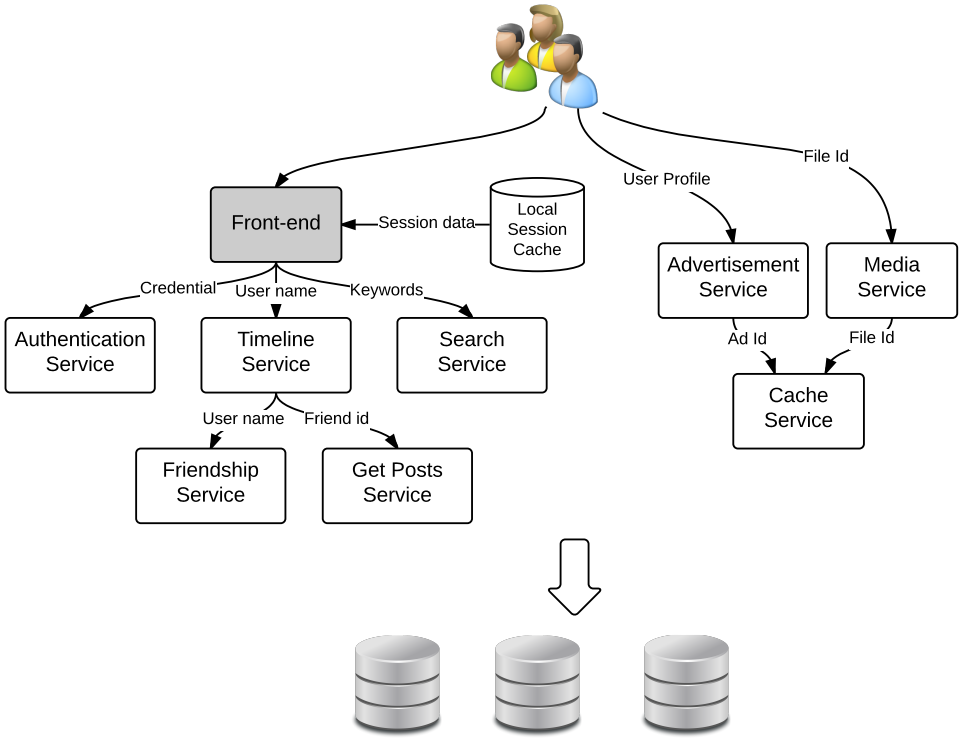


Figure 6.1: A service-oriented social network application

6.3.2 Application Requirements

Our approach requires that requests can be processed by application replicas deployed in other data centers, which involves two important factors: session continuity and data locality.

Session continuity means that the end user should be able to seamlessly interact with the application without losing any internal state even if different data centers process his requests. Stateless applications, such as public knowledge services like Wikipedia and search engine, implicitly satisfy this requirement. For stateful applications, there are ways to make them geographically stateless, such as pushing the states into client side, and session replication across sites [3]. Besides, applications can be divided into multiple stateful and stateless services following the Service-oriented Architecture (SOA) [211]. Figure 6.1 shows an example of a service-oriented social network application. The application first loads the session data of the user and then relays the corresponding data to the underlying stateless services for further processing. The local data center should

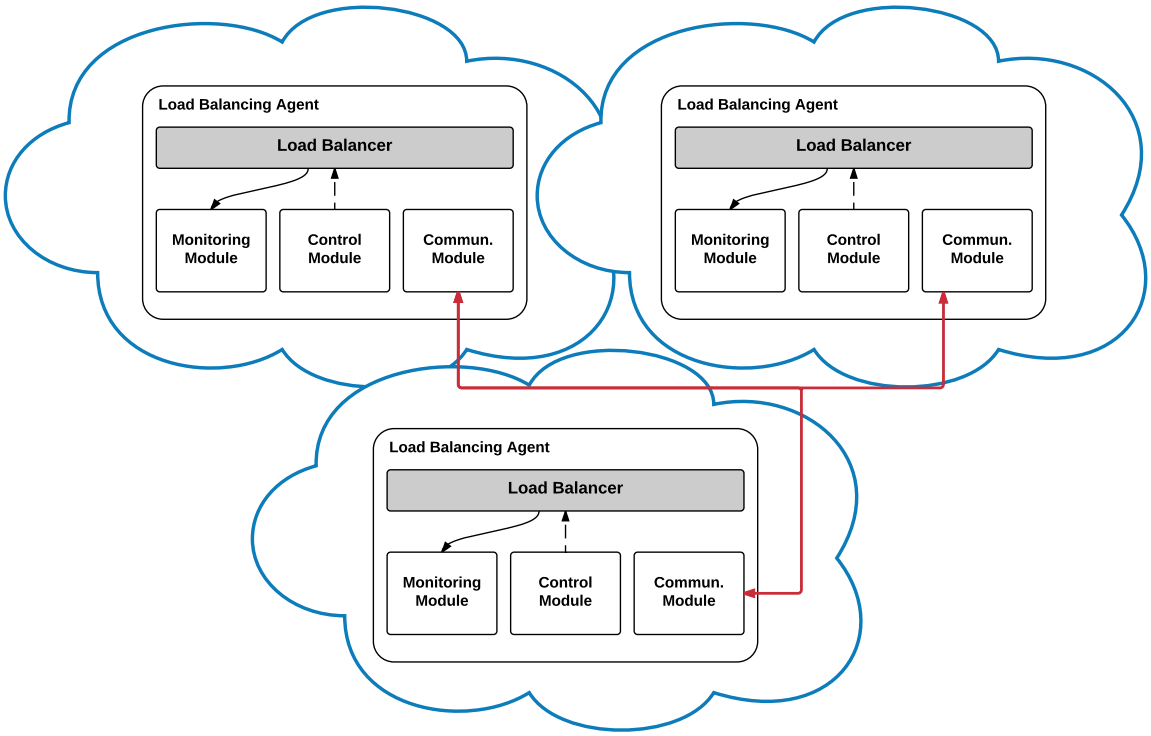


Figure 6.2: Proposed architecture with three data centers involved

always serve the requests for the stateful services. Thus, stateful services cannot be managed by our approach. For these services, they should be sufficiently over-provisioned in preparation for failures and flash crowds. Nevertheless, the majority of the underlying stateless services, which consume the most resources, are eligible to be administrated by our approach.

The second requirement is that persistent data should also be replicated across multiple data centers either partially or entirely as requests can only be forwarded to data centers that have the essential data available. Fortunately, data replication is commonly adopted by applications deployed in multiple data centers nowadays [54, 149], which enhances the applicability of our approach in many scenarios.

6.4 The Proposed Approach

6.4.1 Architecture

Our approach employs a decentralized architecture as shown in Figure 6.2. The load balancing agent is co-located with the application/service load balancer in the corresponding data center to realize fast detection of overload events and perform quick adaptations. They are fully connected with each other through the network to work cooperatively. Each agent comprises the monitoring module (which constantly monitors the incoming requests and the status of the available resources to detect application overload), the communication module (which is in charge of broadcasting its status to other agents and receiving other agents' statuses), and the control module (which quickly adapts the application/service to the detected overload events).

6.4.2 Overload Detection

Choosing the right performance indicator is critical for the detection algorithm. There are several potential indicators we can utilize, such as request rate (request arrival per second), session creation rate (newly created sessions per second), and average response time. In some cases, some indicators cannot truthfully reflect the actual load. For example, the downstream service in an SOA application is usually called by its upstream service using a persistent session, therefore, in this case, session creation rate is not suitable to serve as the overload indicator. In our prototype implementation, we adopt request rate as the indicator since it is more general purpose than other indicators. In addition to the incoming load, the agent also needs to monitor the availability of resources, which can be carried out by periodic health checks.

Another important task is to determine the monitoring frequency. A small monitoring interval enables our approach to timely detect the changes even in the spikiest workload. However, frequent monitoring not only consumes a lot of physical resources, but also causes more false positives. Such behavior can be observed in our experiments results shown in Figure 6.10 and Figure 6.13 with a small amount of requests being rejected by

the admission control approach in the one server down and 245 reqs/s flash crowd settings. Because by using our approach, false positives only result in some of the requests being unnecessarily forwarded to other data centers, instead of being rejected, we choose to favor sensitivity over accuracy and employ a high monitoring rate of every 2 seconds.

With the indicator chosen, we developed a detection mechanism. In the first stage, we profile the machine to determine averagely how many requests per second (c) it can safely handle under the predefined SLA, such as 90% of requests should be served within 1 second. Suppose the requests arrival is a Poisson process and the data center has n machines available for serving requests, the data center is considered overloaded when the incoming workload λ is larger than $n * c + \sqrt{n * c}$ or between $n * c$ and $n * c + \sqrt{n * c}$ for a few consecutive monitoring intervals. The rationality under this approach is that the probability of the result of a Poisson process deviates beyond its standard deviation $\sqrt{\lambda}$ is relatively small, and overload situations often cause much higher load. In this way, we can reduce the amount of false positives caused by highly fluctuant workloads. Note that our framework can employ other detection algorithms as well, such as the one proposed by Kamra et al. [110].

6.4.3 Overload Handling Algorithm

When the overload is detected, the system needs to distribute as many excessive requests as possible to other data centers that have available capacities. Though the response times of the requests that are initially served by the receiving data centers will be negatively affected by the forwarded requests, the SLAs of the receiving data centers can still be ensured as the amount of requests forwarded by our approach should never exceed the remaining capacities available. Our software framework is modularized and can utilize various request distribution algorithms, such as random and greedy algorithms. To get ***optimal*** overall performance, we propose a new distribution algorithm that minimizes the increase of latencies caused by request forwarding. We define the observed latency increase as follows:

$$I(X) = \sum_{i=0}^{n} R_i^{wf}(x_i) - \sum_{i=0}^{n} R_i^{bf} + \sum_{i=0}^{n} F(x_i) \tag{6.1}$$

where x_i is the average amount of requests per second forwarded to the ith data center, $R_i^{wf}(x_i)$ is the latencies observed by all the users initially served by the ith data center with extra x_i requests per second forwarded to it, R_i^{bf} is the total latencies experienced by all the users originally served by the ith data center without extra requests forwarded to it, and $F(x_i)$ is the total latencies felt by the users whose requests are forwarded to the ith data center. Since R_i^{bf} is constant, the latency increase minimization problem is equivalent to:

$$\begin{aligned} \text{minimize} \quad & \sum_{i=0}^{n} R_i^{wf}(x_i) + \sum_{i=0}^{n} F(x_i) \\ \text{subject to} \quad & \sum x_i = N \\ & 0 \leq x_i < S_i \end{aligned} \tag{6.2}$$

where N is the average amount of excessive requests per second to be distributed, and S_i is the maximum average amount of requests per second the ith data center can handle without violating the local SLA.

We model each remote data center as an M/M/1 — processing sharing queue. According to Little's Law, the average response time of the ith data center can be represented as:

$$r_i = \frac{1}{\mu_i - \lambda_i} \tag{6.3}$$

where μ_i is the service rates of the ith data center, and λ_i is the average incoming load to the $i$the data center. Based on Equation 6.3, $R_i^{wf}(x_i)$ and $F(x_i)$ can be respectively modeled as:

$$R_i^{wf}(x_i) = \frac{\lambda_i}{\mu_i - \lambda_i - x_i} \tag{6.4}$$

$$F(x_i) = L_i x_i + \frac{x_i}{\mu_i - \lambda_i - x_i} \tag{6.5}$$

where L_i is the RTT latency between the ith data center and the data center sending the forwarding requests. The optimization function then can be formatted as:

$$\text{minimize} \quad \sum_{i=0}^{n}(\frac{\mu_i}{\mu_i - \lambda_i - x_i} + L_i x_i - 1) \tag{6.6}$$

By removing the constant -1, the optimization problem is transformed to:

$$\begin{aligned} \text{minimize} \quad & \sum_{i=0}^{n}(\frac{\mu_i}{\mu_i - \lambda_i - x_i} + L_i x_i) \\ \text{subject to} \quad & \sum x_i = N \\ & 0 \leq x_i < S_i \end{aligned} \tag{6.7}$$

Since S_i is smaller than $\mu_i - x_i$ as SLA is always stricter than latency at maximum throughput, the optimization function is strictly convex in the feasible space. Besides, the constraints are also convex. Therefore, the optimization problem is strictly convex and can be efficiently and optimally solved using existing convex solvers. According to our experiment results and analysis presented in Section 6.5.8, this problem can be solved quickly enough within milliseconds scale to support instant online decisions. Besides, as long as $N < \sum_i^n S_i$, the feasible set is non-empty, and thus, the problem has a unique global optimal solution since it is strictly convex.

The overall flow of the overload handling algorithm is shown in Algorithm 6. It firstly checks whether the aggregated available capacities of remote data centers can cater all the excessive load (Line 1) and rejects the exceeding requests (Line 2) accordingly. After that, it solves the optimization problem defined in Equation 6.7 and distributes excessive requests among remote data centers (Line 5 and Line 6).

Though rare in probability, simultaneous overloads in multiple data centers can happen. In this case, if the remaining capacities in the rest data centers still can cope all the excessive requests, our approach will serve all the requests. Otherwise, all the data centers will be saturated and our approach will apply admission control to reject the excessive requests in the overloaded data centers.

We treat the network latencies as constants in the optimization problem. However, they often vary dynamically during runtime. Therefore, they need to be dynamically

Algorithm 6: Overload Handling Algorithm

Input: r : the average excessive requests per second
Input: S_i : the maximum average requests per second the ith data center can handle without violating the local SLA
Input: L_i : the latency to the ith data center from the forwarding data center

```
1 if r > ∑_i^n S_i then
2 |   reject r − ∑_i^n S_i requests per second;
3 |   r ← ∑_i^n S_i;
4 end
5 obtain the request distribution plan X by solving the problem defined by Equation
    6.7;
6 respectively forward x_i requests per second to the ith data center;
7 Return;
```

monitored and updated. Sometimes, some data centers can even become disconnected from the network. In this case, they are temporarily removed from the candidate set for request forwarding during the downtime.

6.4.4 Communication Protocol

Load balancing agents in each participating data center communicate among themselves to update their real-time statuses, including their service rates, current loads, and available capacities for offloading. In our prototype, this is implemented through a broadcasting protocol. It makes each agent broadcast its status when its service rate has changed, when its load has varied beyond a predefined percentage, or when some time has elapsed since the last broadcast. Compared to a strategy that broadcasts only in a particular time interval, it not only confines the data error but also makes the application more robust when overload events happen simultaneously in multiple data centers, though such case is expected to be rare. Considering the situation that one agent detects the local application is in an overload condition, according to the protocol, it will immediately inform other agents that there is no available unused capacity offered by it, instead of waiting until the scheduled broadcasting time. This method minimize the chance the agent in another data center happens to be overloaded as well to forward requests to the overloaded

data center, leading it to more severe resource congestion or triggering cascading request forwarding which will incur unnecessary extra latencies.

It is inevitable that sometimes data are not updated timely and causes requests being forwarded to an already saturated data center. In this case, instead of directly rejecting excessive requests in the receiving data center, a cascading request forwarding will be triggered, if it believes there are extra capacities available in other data centers. Because the data center does not distinguish whether a request is originally submitted to it or is forwarded to it by another data center, the forwarded requests this time are formed by a mixture of requests submitted originally to it and requests that have been forwarded once. However, because it is unlikely that more than one data center fall into overloaded situations nearly at the same time, such scenarios will cause limited impact.

6.4.5 Prototype Implementation and Deployment

Since the target of the proposed approach is to detect and handle application overload as soon as possible, it is preferable to develop it as a part of the load balancer so that it can react instantaneously after the detection of the overload events. However, state-of-the-art load balancers, such as HAProxy 1.6 [83], do not support to program such complex configuration. Therefore, we implemented the agent as a separate program. Fortunately, as some of the load balancers already have built-in monitoring and health check tools, we still can utilize them to ease the implementation and deployment of our approach.

The implementation follows the architecture shown in Figure 6.2. In the implementation, we use HAProxy 1.6 as the load balancer and rely on it to monitor the performance indicators and check the machines' health. The agent is written as a separate Java application. Its monitoring module periodically fetches the monitored information through HAProxy's stats console in CSV format. Then it extracts the required performance indicators and health statuses of the attached servers and passes them to the overload detector. The overload detector then uses the overload detection algorithm to judge whether the application is overloaded. In the case that application overload is detected, the control module configures the load balancer to adapt to the load based on the proposed overload handling algorithm. The Java agent program should be collocated with the HAProxy

```
listen webcluster
    bind    *:80
    mode    http
    option  httplog
    stats   enabled

    balance roundrobin
    option  httpchk HEAD / HTTP/1.0
    option  forwardfor

    acl monitoring src localhost
    acl test rand(250) lt 33
    tcp-request content reject if test !monitoring

    server local-1 172.31.61.40:80 check inter 2000ms weight 35
    server local-2 172.31.54.227:80 check inter 2000ms weight 35
    server local-3 172.31.55.239:80 check inter 2000ms weight 35
    server local-4 172.31.61.130:80 check inter 2000ms weight 35
    server local-5 172.31.58.35:80 check inter 2000ms weight 35

    server ireland 52.208.134.152:80 weight 13
    server tokyo 54.132.154.120:80 weight 29
```

Figure 6.3: An example of the dynamically generated HAProxy configuration

load balancer to minimize network latency.

Both request forwarding and admission control are implemented by dynamically changing the configuration of the HAProxy load balancer. In detail, the control module dynamically generates a new configuration file for the HAProxy when it is necessary to perform changes during runtime. The configuration change is performed through a script which automatically reloads the new configuration to the running HAProxy process.

The request forwarding mechanism is implemented by dynamically adding the IP addresses of the load balancers located in the remote data centers to the local load balancer's configuration file as normal servers. The number of forwarded requests is dynamically adjusted by assigning relative weights to the servers and remote data centers using the weighted round robin algorithm supported by HAProxy. The relative weights are obtained by solving the load distribution problem defined in Equation 6.7 using the Primal-Dual Interior-Point method built in the JOptimizer solver [105].

We take advantage of the Access Control List (ACL) mechanism in HAProxy to implement admission control. It is traditionally used to define the white list and black list of IP addresses to prevent abusing. Our agent utilizes it in a different manner. We define an ACL as a Bernoulli trial instead of a fixed list. In this way, the incoming request will

be served if the random test result is successful; otherwise, it will be rejected by the load balancer. Note that in a production environment, instead of directly rejecting requests; a better approach is to allow the load balancer to reply a customized error page or a default page, which is already supported by HAProxy.

Figure 6.3 shows an example of the dynamically generated HAProxy configuration by our approach. In line 31, it defines an ACL called monitoring, which is used to prevent monitoring requests issued by the Java agent co-located in the same machine to be rejected by the admission control mechanism. Lines 32 and 33 specify the admission control configuration in which the load balancer will randomly drop 33 out of 250 incoming requests. The servers starting with "local" locate in the current data center. They are health checked every 2000ms as shown in the configuration, and the load balancer talks to these servers through their internal IPs. The last two lines specify the request forwarding settings to the remote data centers located in Ireland and Tokyo. They receive forwarded requests through the public IPs of their load balancers.

The communication module and its protocol are implemented using Java sockets over persistent TCP connections. Each Java agent constantly maintains a connection to all the other known agents and continually listens to the updates sent by them.

6.5 Performance Evaluation

We evaluated our prototype implementation on Amazon Web Services IaaS infrastructure located in North Virginia, Ireland, and Tokyo. We first introduce the benchmark application and the experimental testbed. After that, we describe the workloads we used for experiments. Finally, we explain each experiment and present the results.

6.5.1 Benchmark Application

We used the Wikibench benchmark tool [203] as the testing application. The benchmark tool consists of three components:

- a stateless web application server installed with the MediaWiki application [140] —

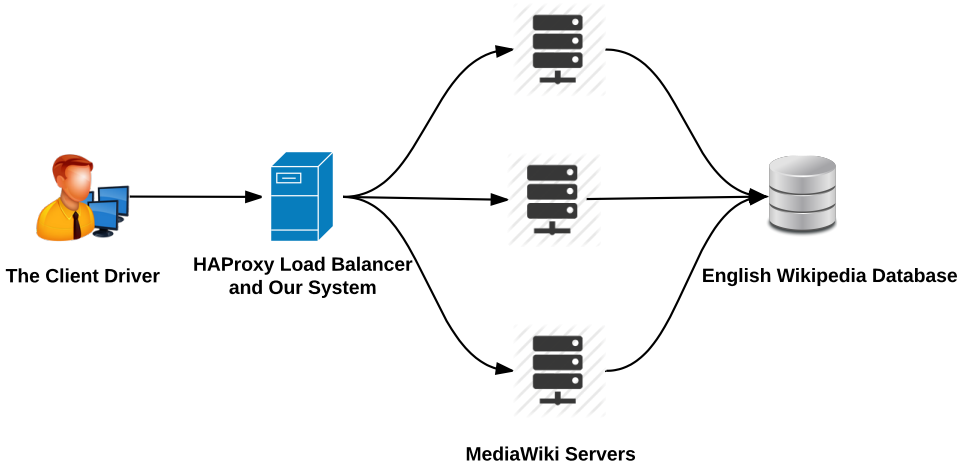


Figure 6.4: The architecture of the benchmark application

an open source version of Wikipedia.

- a MySQL database loaded with the English Wikipedia pages by January 2008.
- a client driver that mimics the behavior of users by sending requests to the MediaWiki server according to the given workload.

The reason we chose this benchmark is that it is stateless, which fits our prerequisite. Because our focus is on the application tier, to allow deploying a cluster of application servers, we put an HAProxy load balancer before the servers and changed the frontend configuration of the MediaWiki application servers to the IP address of the load balancer. As stated before, the load balancing agent is deployed along with the HAProxy load balancer on the same machine. Figure 6.4 demonstrates the architecture of the benchmark application.

6.5.2 Experimental Testbed

We set up the experimental testbed in three data centers owned by Amazon Web Services: US-east1 North Virginia, EU-West1 Ireland, and AP-Northeast1 Tokyo. Table 6.1 lists the RTT latencies between North Virginia and the other two data centers tested using ping. In the experiments, we needed to emulate resource failures and flash crowds in one data center. We chose North Virginia data center as the data center that experienced failures

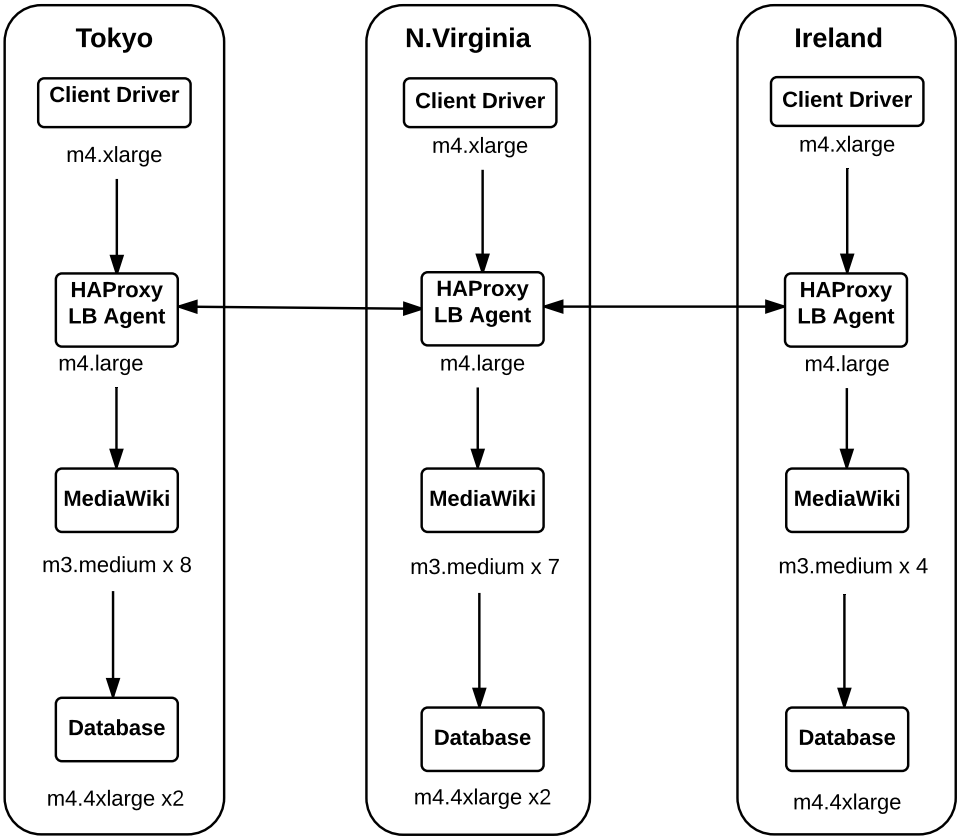


Figure 6.5: The experimental testbed

Table 6.1: Latencies between data centers in milliseconds

	Ireland	**Tokyo**
North Virginia	76.3	167.2

and flash crowds. Besides serving their loads, the other two data centers accepted loads directed from North Virginia data center when overload occurred to it.

The detailed experimental testbed is illustrated in Figure 6.5. In each data center, we deployed one client driver using the m4.xlarge instance, and one HAProxy server along with the load balancing agent running on an m4.large instance. We respectively launched eight and four m3.medium instances in Tokyo and Ireland acting as application servers. The North Virginia data center, in the meantime, is equipped with seven application instances. Besides, to ensure that the data layer did not become the bottleneck, we launched different numbers of database instances to cope the load in each data center.

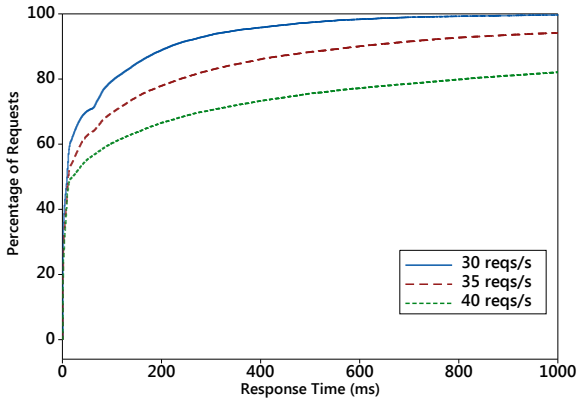


Figure 6.6: CDFs of the profiling tests with different average workload rates

6.5.3 Workload

We used synthetic workloads generated according to real requests submitted to the English-language edition of Wikipedia in September 2007 [204] to test our approach. We first performed profiling tests to determine on average how many requests one m3.medium application server can handle without violating the SLA and its service rate. We defined the SLA as 90% requests should be replied within one second. We assume the workload arrival is a Poisson process and follows the exponential distribution, which is indicated by the literature [41]. This can be justified by the fact that each request is submitted independently and occurrence of each request does not affect the probability that a second request will occur. Base on this assumption, we respectively created three workloads with an average rate of 30, 35, and 40 requests/s.

Figure 6.6 depicts the cumulative distribution functions (CDF) of the response times obtained from the profiling tests. It shows that 35 requests/s is the largest amount of workload an m3.medium instance can handle without violating the SLA. Furthermore, we respectively calculated the service rates of the three tests according to Equation 6.3. Then we averaged them to obtain the estimated service rate for one instance, which is 41 requests/s.

In the following experiments, we assigned 35 requests/s unused capacities in Virginia and Ireland, and 70 requests/s unused capacities in Tokyo. According to the profiling results and the capacity setting, we generated synthetic workloads for the following ex-

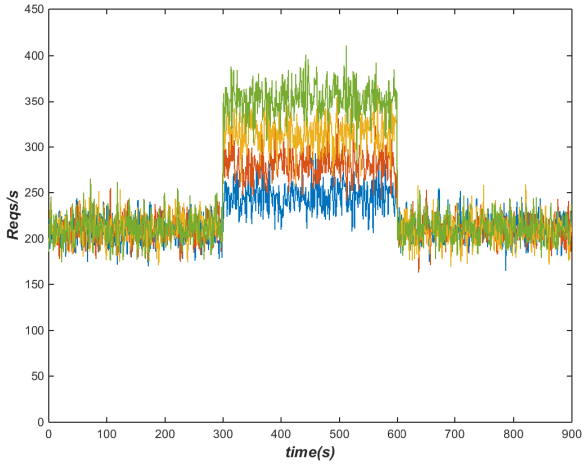


Figure 6.7: The workloads with flash crowds range from 117% to 183% of the normal load

periments. We first generated the background workloads for the Ireland and Tokyo data centers respectively with average incoming rates of 105 requests/s and 210 requests/s. To test the performance of the approach during resource failures in North Virginia data center, we generated a workload with average request rate of 210 requests/s. For flash crowds cases, we created four workloads with different levels of flash crowds as shown in Figure 6.7. Each workload experiences a total five minutes of flash crowd. The peaks of the flash crowds range from 117% to 183% of the normal load. All the workloads last for 15 minutes and suffer either server failures or flash crowds starting from the 300s time point for 300 seconds. We particularly chose the length of 300 seconds because it is the default value of the waiting time for server booting in Amazon Auto-scaling Service. In this way, we can emulate the situations that a commercial auto-scaler solely manages the application and demonstrate its resulted application performances during the overloading periods.

6.5.4 Benchmarks

To test the performance of our prototype, we compare our approach with the following two benchmarks:

- **Request queueing**: the first benchmark queues up all the requests in the local servers and imposes no admission control when the application is overloaded. It

mimics the situation that the auto-scaler is booting new servers while the requests are queued up in the local servers.

- **Admission control**: the second benchmark directly imposes admission control when the application is overloaded. It emulates the case that the auto-scaler asks the load balancer to reject excessive requests while it is booting new servers.

To test the performance of the request forwarding algorithm, we compare with the following two greedy algorithms:

- **Greedy**: it always forwards possibly maximum amount of excessive requests to the data center with largest available capacity one by one.

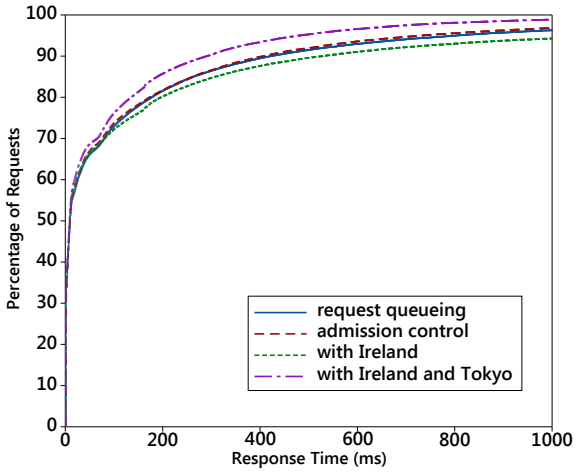
6.5.5 Performance under Resource Failures

In the first set of experiments, we performed tests using our approach and the benchmarks in resource failure situations. In the experiments, we purposely removed some machines from the load balancer at 300s time point to create failures and then added them back to the load balancer after 5 minutes to mimic recovery from failures.

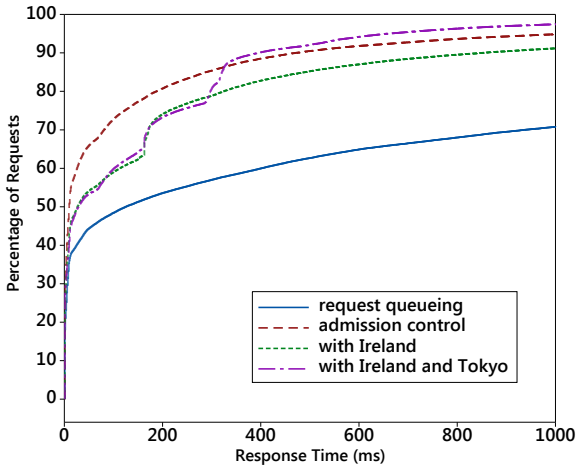
We ran our approach with two configurations. In the first configuration, we only utilized the unused capacity in the Ireland data center. In the second configuration, we considered unused capacity in both Ireland and Tokyo data centers. Figure 6.8 shows the CDFs of our approach and the two benchmark approaches respectively during the failing periods under different numbers of server failures[2].

Without proper overload handling mechanisms, the application in the North Virginia data center suffered severe performance degradation when requests were queued up, and it became completely unresponsive in the case of five server failures. If we added simple admission control mechanism, the application performance was able to be maintained within acceptable level at the cost of rejecting plenty incoming requests. As shown in Figure 6.8, we can observe one and two shoulders in the CDFs of the two settings. This phenomenon was caused by the increased network latencies incurred by the request

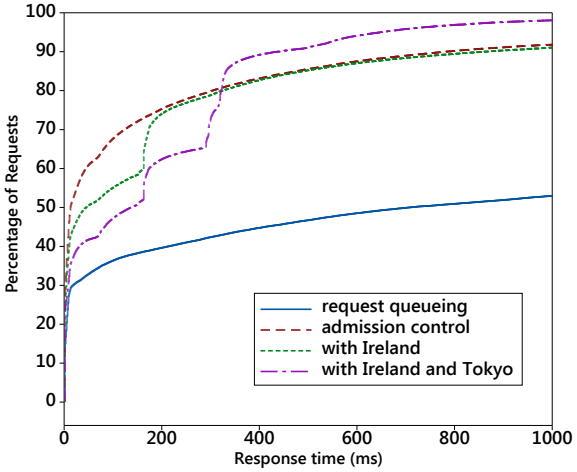
[2]In the reported results, rejected requests are not counted in the CDF graphs.



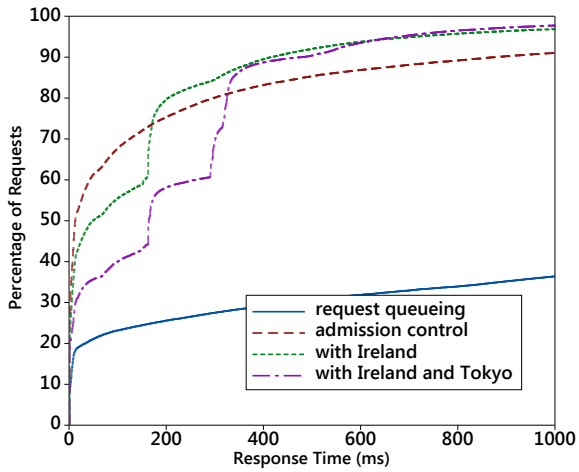
(a) one server failure



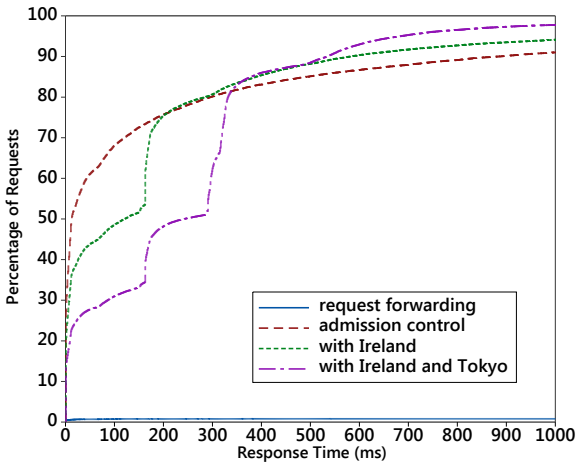
(b) two server failures



(c) three server failures

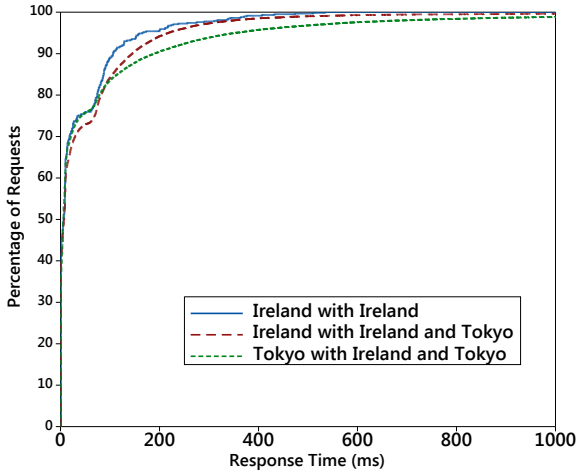


(d) four server failures

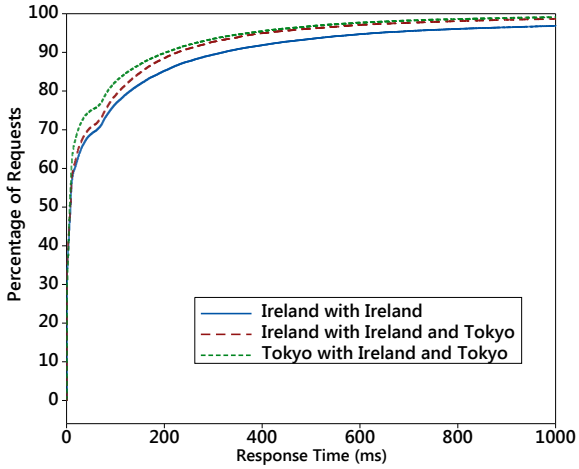


(e) five server failures

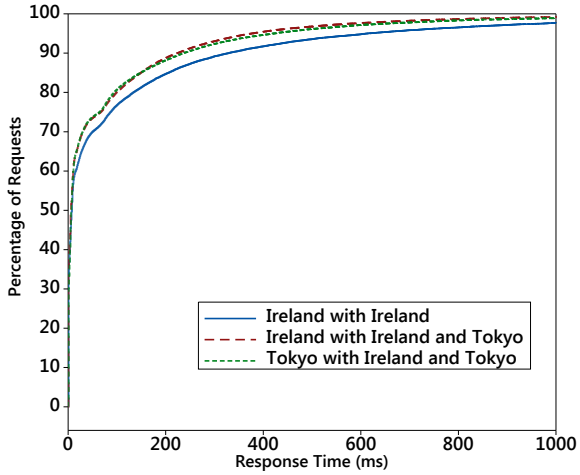
Figure 6.8: CDFs of the North Virginia data center during the failing periods with different approaches



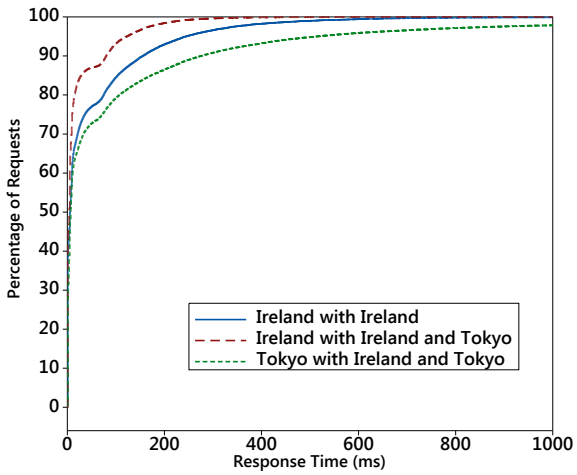
(a) one server failure



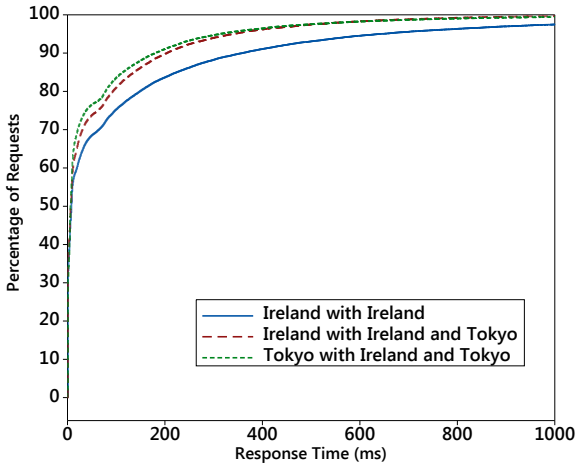
(b) two server failures



(c) three server failures



(d) four server failures



(e) five server failures

Figure 6.9: CDFs of the data centers receiving forwarded requests during failing periods

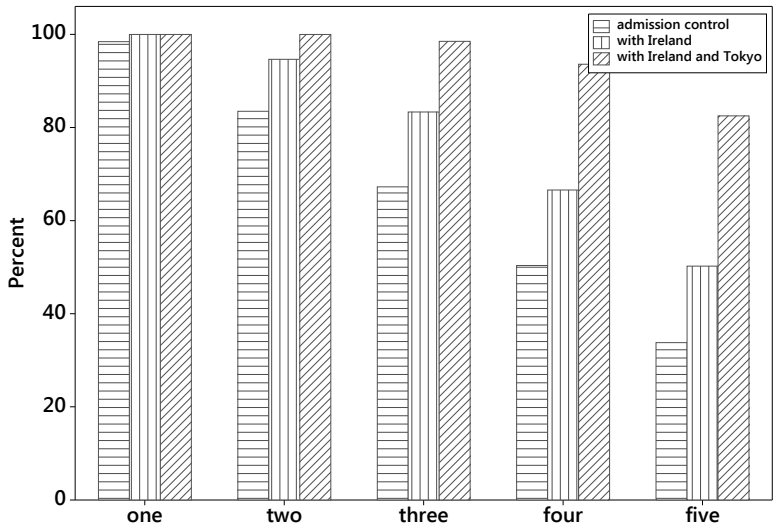


Figure 6.10: Percentage of admitted requests during failing periods

forwarding mechanism. In our experiments, though request forwarding increased the latencies of some requests, it did not result in the SLAs being violated. Comparing to the queuing delays when no overload handling mechanism is in place, the additional network latencies incurred by request forwarding are still acceptable as long as the latencies between data centers are moderate. Our overload handling algorithm is encouraged to forward requests to data centers that are close to the originating data center as it aims to minimize the overall latency increase.

In addition to the failing data center, we also evaluated the performances of data centers that received the forwarded requests. Figure 6.9 presents the CDFs of the data centers receiving the forwarded requests when failures were happening in North Virginia data center. In all cases, the SLA was strictly honored because of the constraints on the amount of forwarded requests the remote data centers can serve in the optimization problem.

Furthermore, comparing to just using admission control, our approach was able to increase the number of served requests. Figure 6.10 lists the proportions of the admitted requests during the failing periods in the corresponding experiments. It shows that the power of request forwarding depends on the amount of unused capacity available in other data centers. As shown in Figure 6.10, the configuration utilizing the capacity of both Ireland and Tokyo data centers can serve more requests than the configuration

that used capacity only in Ireland data center. When applying admission control only, a small portion of requests was rejected when one server was down even though the local capacity should be able to handle it, which was caused by false alarms generated by the overloading detector. While using our approach, these requests were still served by remote data centers.

6.5.6 Performance under Flash Crowds

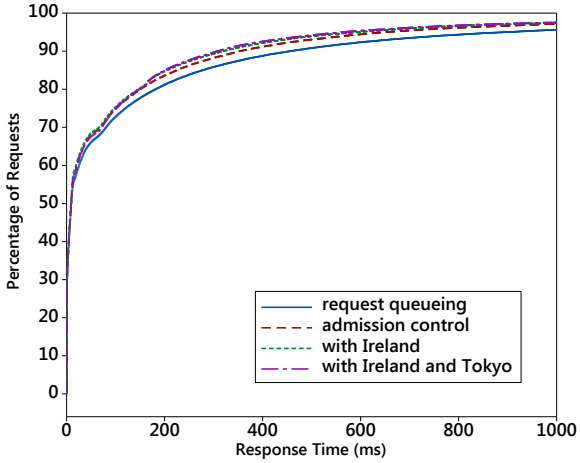
We tested our approach under flash crowd situations using the workloads depicted in Figure 6.7. We utilized the same settings of Section 6.5.5. The resulted CDFs for the baselines and our approach for North Virginia data center that was under flash crowds are delineated in Figure 6.11. The corresponding CDFs for the receiving data centers are presented in Figure 6.12. The percentages of admitted requests during the flash crowd periods are compared in Figure 6.13.

The experiments show similar trends as the experiments in resource failure situations. Nevertheless, the impact of short-term overload on the application performance in the flash crowd experiments was not as severe because the extra load can be directed to more servers. For the same reason, more percentages of requests were served by the application.

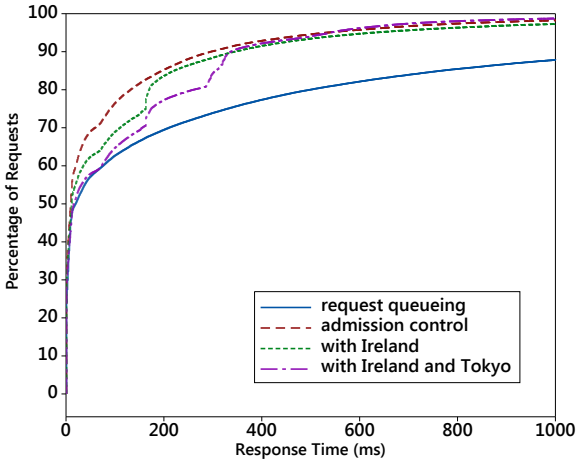
6.5.7 Performance of the Request Forwarding Algorithm

We utilized the same testing platform to evaluate our request forwarding algorithm (specified as Min Latency Increase in the results), except we employed a workload with 70 requests/s for the North Virginia data center and considered all those requests were excessive requests need to be forwarded. In this way, we can eliminate the impact of the requests served by the North Virginia data center, which is not in our optimization target, to the results.

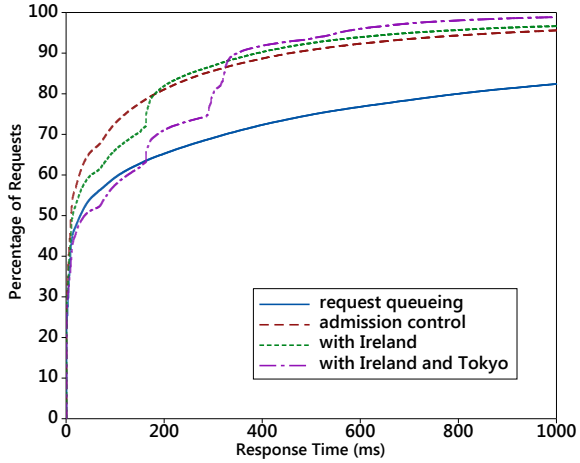
Figure 6.14 shows the results of our algorithm compared to the Greedy algorithm for the aggregated requests of the forwarded requests and the requests originally served by the remote data centers. Our algorithm is able to outperform the Greedy algorithm in



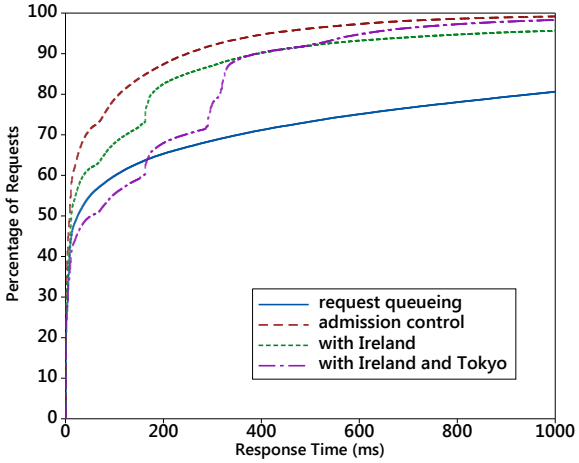
(a) 245 reqs/s flash crowd



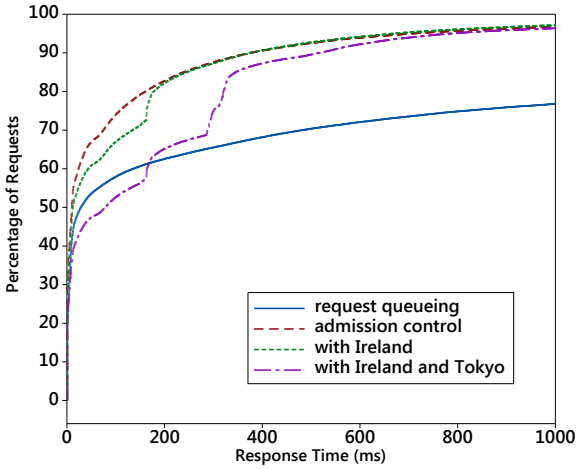
(b) 280 reqs/s flash crowd



(c) 315 reqs/s flash crowd

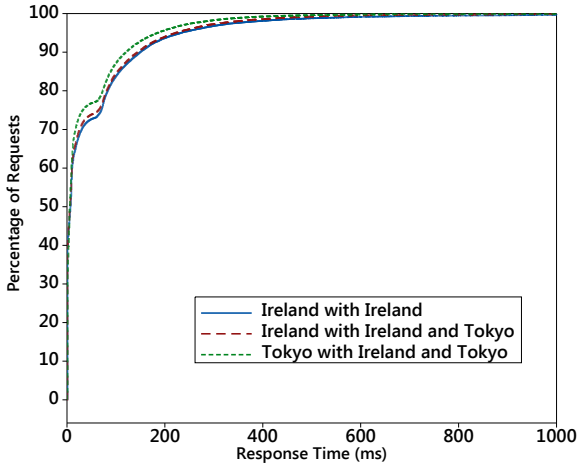


(d) 350 reqs/s flash crowd

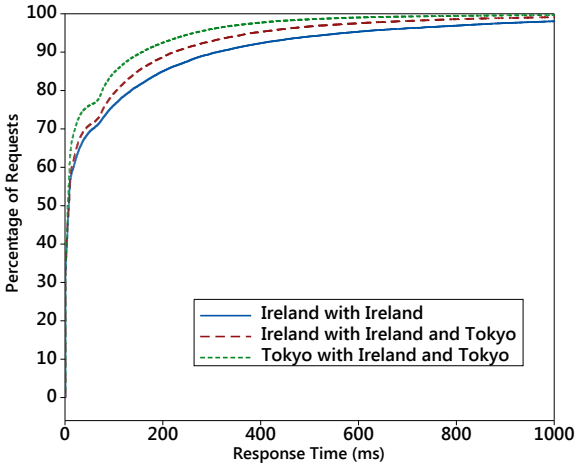


(e) 385 reqs/s flash crowd

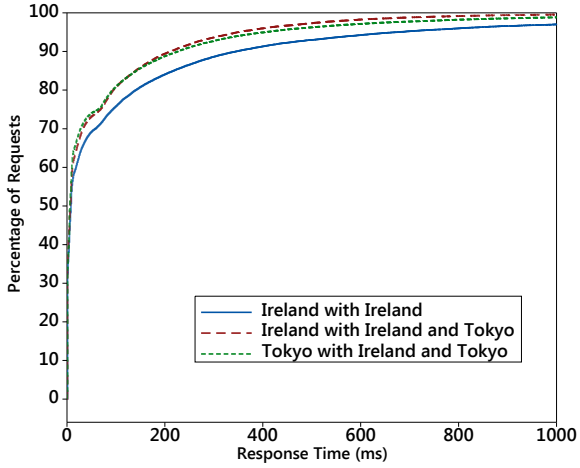
Figure 6.11: CDFs of the North Virginia data center during the flash crowds with different approaches



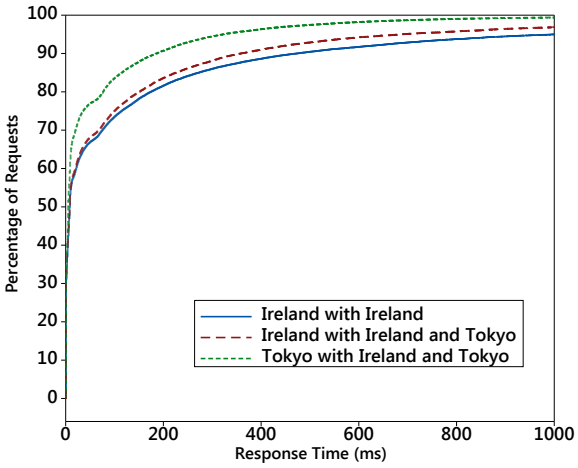
(a) 245 reqs/s flash crowd



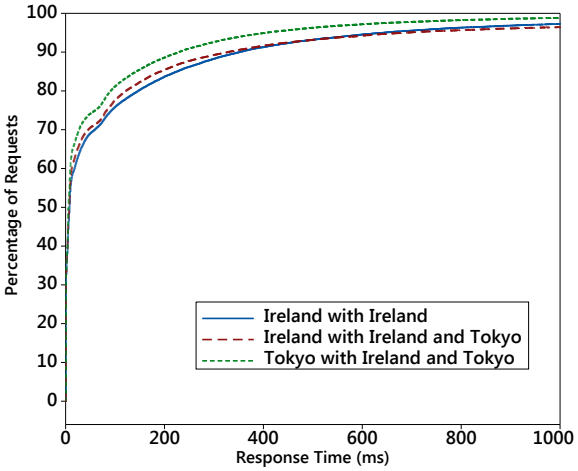
(b) 280 reqs/s flash crowd



(c) 315 reqs/s flash crowd



(d) 350 reqs/s flash crowd



(e) 385 reqs/s flash crowd

Figure 6.12: CDFs of the data centers receiving forwarded requests during flash crowds

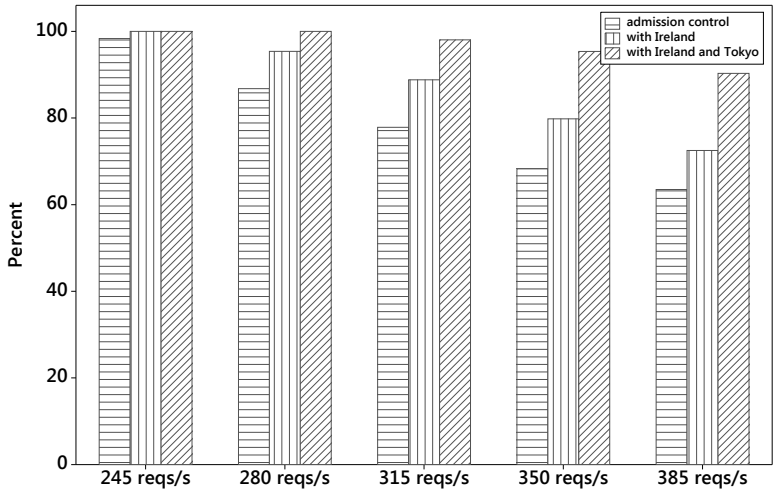


Figure 6.13: Percentage of admitted requests during flash crowd periods

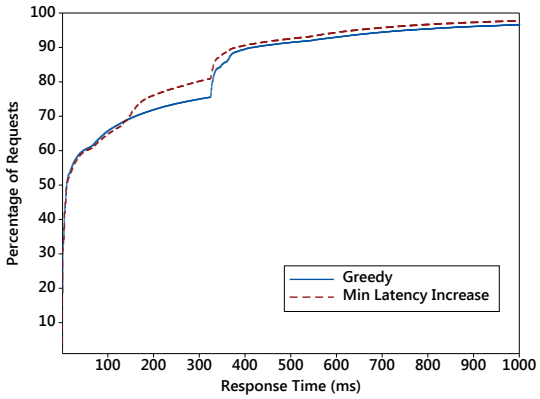


Figure 6.14: The performance of algorithms on the aggregated requests

our experimental setting, because the Greedy approach overlooks the longer network distance traveled by the forwarded requests.

6.5.8 Algorithm Scalability

Our approach requires an efficient solution for the workload distribution problem during runtime. In this experiment, we demonstrate that this problem can be tackled very fast by a convex solver even when a large number of data centers are involved.

Since the input to the problem consists of the statuses of the data centers, the latencies to the data centers, and the amount of the excessive load, the algorithm complexity is dominated by the number of involved data centers. We randomly generated 100 prob-

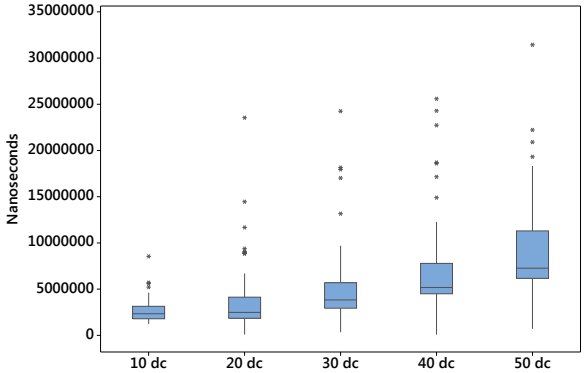


Figure 6.15: Measured running time for solving the workload distribution problem

lems with specific numbers of data centers ranging from 10 to 50 and measured their running time on a desktop equipped with 8 core CPUs and 16 GB of RAM. Results are presented in Figure 6.15. Even in the worst case with 50 data centers, the runtime was below 35ms, which is acceptable for making real-time decisions. In reality, the deployment usually involves less than a handful data centers, and the algorithm imposes negligible overhead in these cases.

6.6 Related Work

6.6.1 Overload Management

There have been plenty of works that aims to tackle overloads caused by failures and flash crowds using Cloud resources. However, all of them differ from our work in their target or approaches.

The approach that is commonly adopted by the industry and is intensively researched is auto-scaling [133]. It relies on dynamically provisioning new resources to meet the resource scarcity. Some of the auto-scaling works have been focusing on how to predict the future workloads and provision enough resources in advance [58, 62, 87, 94, 100, 104, 125, 164]. Other approaches provision resources reactively either after detecting the overload events [53, 69] or when the utilization has reached a certain threshold.

As stated before, resource failures and some flash crowds are often unpredictable, and it takes the auto-scaler considerable time [138] to provision new resources. Therefore, an

auto-scaler alone cannot adequately deal with these situations. Our work can fill in this gap for applications deployed in multiple data centers by supplementing and enhancing state-of-the-art auto-scaling solutions.

In the previous chapter, we proposed an auto-scaler using unreliable spot instances for web applications. It relies on sufficiently over-provision the application to counter the terminations of spot instances. The proposed approach can be an ideal partner for it. By working cooperatively with it, the spot-based auto-scaler can either reduce the amount of over-provisioned resources to reach the same level of protection or elevate the reliability of the application using the same amount of over-provisioned resources.

Cloud burst is a term often used in hybrid Cloud settings referring to dynamically provisioning resources in Cloud either to accelerate execution or to handle flash crowds when the local facility is saturated. Many approaches have been proposed to realize this vision [25, 97, 218, 222]. In a sense, they are similar to our work as they also forward requests to remote data centers. Except that, they are more close to the auto-scaling approaches as their major focus is on how to provision and deprovision resources in Clouds to meet the workload demand while our approach aims to manage short-term workload distribution.

Another possible method to reduce the impact of overload is to enforce admission control. Chen et al. [38] proposed a flash crowd detection and mitigation approach based on application-level measurements and admission control. In addition to protecting the application server from crashing, their target also covers protecting the network from being congested. However, in Clouds, since the provider offers strong data center network and high incoming and outgoing bandwidth, this is not a concern. Different to their system, our approach uses both request forwarding and admission control as the last line of defense to protect the application from performance degradation and crashing.

Chandra and Shenoy [33] researched using dynamic resource allocation among different applications in a data center to cope with flash crowds. Different to them, our work addresses the overload management problem from an application provider's perspective instead of from an infrastructure provider's angle. Regarding applications that are composed of multiple components, Gandhi et al. [70] and Klein et al. [118] explored bor-

rowing resources from components that have the available capacity or can be terminated temporarily to support the core services under flash crowds.

6.6.2 Geographical Load Balancing

Geographical load balancing has been applied to tackle different challenges. Commercial DNS Load Balancer, such as Amazon Route 53 [14] enables application providers to direct their customers to different data centers according to their location and other factors, such as energy consumption and carbon footprint [131]. However, such technique is not suitable to our needs as it takes some time to populate the DNS settings across the layered DNS servers and it is impossible to realize fine-grained control over the traffic flow.

Centralized geographical load balancing solutions gather all the user requests and then distribute them among data centers. They are widely developed for saving energy and carbon footprint [143, 225]. This architecture is also not applicable to reach our goal as it incurs extra network latency to every request and reduces the benefit of deploying the application on multiple geographically distributed Clouds.

Grozev and Buyya [78] proposed an approach that dispatches users to the underlying data centers at the entry point of the application framework according to the regulation requirements and the available resources in each data center. As the client only talks to the entry point at the start of the session, its impact on user experience is minimized compared to the centralized solutions. On the other side, this approach limits its capability to control the load on each data center accurately.

Cardellini et al. [31] explored using DNS as the first layer of request distribution and centralized or decentralized request forwarding as the second layer to balance the workload among web nodes.

Our solution is different to the methods as mentioned earlier. We adopt a decentralized architecture composed of individual load balancing agents deployed in each participating data center to balance the extra load. The agent is only activated temporarily when an overload occurs; hence, requests under normal situations can always be served by the closest data centers, and no extra network latency is introduced. Andreolini et al. [15] also proposed a decentralized autonomic system to handle the overload situa-

tions. However, their target is to reduce the performance variability instead of serving as many requests as possible under SLA requirements. Ardagna et al. [18] utilized both request redirection and resource allocation to manage resources. Their aim is to minimize resource cost under SLA constraints.

In other domains, Ranjan [159] and Alzoubi et al. [7] employed request redirection techniques, such as anycast, to improve the performance of content delivery networks.

6.7 Summary

In this chapter, we proposed an approach that supplements and enhances state-of-the-art auto-scalers for applications deployed in multiple Clouds. It is capable of quickly react to short-term overloads occurred to any participating data center by temporarily forwarding the excessive requests to data centers with available capacities according to our proposed optimal workload distribution algorithm, and enforcing admission control as the last line of defense. Our approach adopts a decentralized architecture that deploys a load balancing agent within each data center. The agent monitors the local application, detects overload events, and quickly adapts to them according to real-time resource availability in other data centers to minimize their impact on the application performance. We implemented a prototype and evaluated it across AWS's US, Europe, and Asia data centers. The obtained results show that our approach can quickly detect overload situations caused by either resource failures or flash crowds and is effective in improving application performance during resource contention periods.

From Chapter 3 until this Chapter, we have introduces the contributions proposed in this thesis. In the next chapter, we summary them and propose some future directions.

Chapter 7
Conclusions and Future Directions

In this chapter, we summarize the research works proposed in this thesis about the areas of deployment and provision of web applications in distributed computing Clouds. It sums up the contributions and identifies some future research directions to pursue in these fields.

7.1 Summary and Conclusions

CLOUD Computing allows users to acquire resources according to their real-time demand from Cloud data centers in a pay-as-you-go manner, which eliminates the needs for users to maintain local infrastructures and enables them to focus on their core business. Due to its appealing features, it has been attracting web application providers to migrate and develop their systems onto it.

Many application providers have deployed their applications across multiple geographically distributed Cloud data centers. In addition to utilizing a single data center, it brings extra advantages like 1) better QoS, 2) higher availability, 3) regulation compliance, 4) avoidance of vendor lock-in, and 5) cost-efficiency. However, managing web applications in a multi-Cloud environment face some remaining challenges.

In this thesis, we divided the challenges into two primary aspects: 1) deployment of applications on distributed computing Clouds, and 2) provision resources to meet the QoS requirements. We have conducted a thorough literature review, proposed solutions, and implemented prototype systems to improve the current state-of-the-art in these fields. In particular, Chapter 1 described this thesis' objectives in more details and

This chapter is partially derived from: **Chenhao Qu**, Rodrigo N. Calheiros, and Rajkumar Buyya. "Auto-scaling Web Applications in Clouds: A Taxonomy and Survey", *ACM Computing Surveys*, 2016 (under review).

highlighted its main contributions. It also presents the motivation and structure of the thesis.

In Chapter 2, we performed a literature review of previous researches regarding three major problems that are tackled in this thesis. Firstly, we surveyed and classified works that aim to identify satisfactory Cloud services according to users QoS requirements. Then we discussed proposed approaches to select geographically dispersed data centers among candidates to host web applications. In the last part, we focused on the auto-scaling problem for web applications in Clouds. We presented a comprehensive taxonomy of auto-scaling techniques and compared existing systems based on that. For each problem, we also defined its scope and identified the significant challenges need to be addressed.

Chapter 3 and Chapter 4 focused on the deployment aspect of managing web applications in distributed computing Clouds.

To realize Cloud service discovery regarding users' individual QoS requirements, Chapter 3 presented a technique that uses hierarchical fuzzy inference system to evaluate and rank the satisfactory degree of the Cloud services to users' individual requirements. It is compatible with the hierarchical Service Metric Index of Cloud services [48] defined by the Cloud Service Measurement Initiative Consortium (CSMIC). It also allows users to flexibly specify their requirements using either numerical values or vague linguistic terms. Furthermore, users can describe their preferences among metrics with fuzzy linguistic hedges. We demonstrated its efficacy through simulation experiments and case studies.

Chapter 4 proposed approaches that select the data centers to host web applications with strong inter-data center consistency requirement to minimize violations of the Service Level Objectives (SLO), and optimizes the selection under a dynamic workload. We modeled response latency observed by the end users. Within the model, we proposed sub-models to estimate the latency caused by consistency protocols of Cassandra and Galera Cluster databases. A genetic-based algorithm was devised to efficiently solve the proven NP-hard problem to select the best data centers. Based on that, two heuristic algorithms that consider both SLO violations and migration cost were proposed to optimize

the selection when workload has changed. We have conducted simulation experiments to demonstrate that it is beneficial to deploy applications in geographically dispersed data centers even it requires strong inter-data center consistency, and it is essential to take the extra latencies caused by the database consistency mechanism into account when selecting the data centers. We have also illustrated through experiments that our approach is more effective compared to the greedy algorithm and much more efficient than the exhaustive algorithm.

Chapter 5 and Chapter 6 targeted to solve the issues in resource provision of web applications in Clouds.

To improve the cost-efficiency of resource provisioning for web applications in Clouds, we proposed an auto-scaler in Chapter 5. In addition to the previous approaches, it is not only capable of provisioning and deprovisioning resources dynamically according to real-time workload but also can achieve significantly more cost saving. It utilizes a mixture of homogeneous on-demand VMs and heterogeneous unreliable spot instances to achieve both high availability and considerable cost saving. We proposed a fault-tolerant model to handle the sudden termination of spot instances. We also designed scaling algorithms that are compatible with the fault-tolerant semantics. We implemented a prototype for Amazon AWS and a simulation tool based on CloudSim [30] for determining the proper configurations and enabling repeatable evaluations. The intensive simulation studies and experiments on Amazon AWS confirmed the efficacy of our approach.

Chapter 6 presented a decentralized geographical load balancing technique to handle short-term overloads that are complicated to be dealt with auto-scaling technology alone for applications deployed in multiple data centers. The proposed approach deploys a load balancing agent in each data center, which monitors the incoming workload and the status of available resources to detect overload timely. In case overload is detected, it forwards part of the workload to other data centers that have unused capacities and enforces admission control as the last line of defense when there is no more capacity available. We also proposed a convex model to quantify the overall latency increase caused by request forwarding and used a convex solver to efficiently and optimally determine how much workload should be forwarded to each remote data center when overload happens. We

Figure 7.1: Future Directions

implemented a prototype and evaluated it on Amazon's global infrastructure across US, Europe, and Asia. The obtained results compared with a queuing approach and an admission control approach demonstrated its effectiveness and feasibility.

7.2 Future Directions

The thesis spans across the deployment and provision aspects of managing web applications in distributed computing Clouds. In each subfield, there are promising future directions that can be explored. Figure 7.1 presents an overview of the future directions we propose.

7.2.1 Cloud Services Discovery

Regarding Cloud services discovery, the following paths can be pursued.

Services Discovery for Complex Applications

Currently, the Cloud service discovery tools only evaluate satisfiability of services for single VM requests. For applications that are composed of multiple components, application providers need to individually find the suitable services for each component and intersect them to identify a list of providers that can provide satisfactory services for each component. Besides inconvenience, this also could cause suboptimal selection decisions as this approach puts too much weight on satisfiability of individual component and fails to consider the application as a whole. Therefore, methods that can comprehensively evaluate the satisfiability of the entire application is desirable.

Services Discovery with Quantified Variability

In our approach, we used statistical indicators like mean, median, and first quartile as the values for performance. In addition to that, metrics like standard deviation, and square root errors can be used to quantify the variance of performance and be considered in the evaluation process. New methods need to be proposed to suitably balance the weights of these metrics according to user requirements.

7.2.2 Clouds Selection and Optimization

In the area of selecting Cloud data centers, the following directions have not been explored.

Selection and Optimization for Applications with Strong Consistency and Partial Replication

The technique proposed in Chapter 4 assumes that data are replicated in each data center. Though it is commonly adopted, it is certainly cost-inefficient. Approaches are in need to support the deployment of applications with both strong inter-data center consistency requirements and partial data replication, which calls for not only new latency models but also efficient algorithms that holistically determine the locations of data centers and the placement of data.

Optimization with Fine-Grained Migration Cost Model

We used the number of migrations needed to be conducted as the metric for migration cost, which is coarse-grained, as the cost of migration is affected by many factors, such as application itself, data volume, bandwidth cost, and labor cost. A fine-grained model of migration cost can help application providers to determine better whether to adjust the deployment or not and how the changes should be conducted based on quantitative analysis.

Reliability and Disaster-Aware Selection

One significant advantage of utilizing multiple data centers is that it provides high availability in case of data center outages. However, if the chosen data centers are not distant enough, massive catastrophic natural disasters or blackouts caused by complete grid failures still may disrupt the service altogether. Therefore, the selection algorithms should take reliability factors into account to ensure the applications are available even under extreme situations. For example, the algorithm should not select data centers that share

the same power source and Internet service provider, or data centers may be affected by the same storm and earthquake.

7.2.3 Auto-scaling

In regards of auto-scaling, we list the potential future directions as follows.

Holistic Auto-scaling for Service-Oriented Applications

The research on scaling complex service-oriented applications are still at early stage and limited literature can be found in this area. Moreover, due to lack of accurate resource estimation models, only a simple approach that tentatively and recursively provision resources to a selected service is proposed [98], which takes a long time to reach the overall target performance. If accurate resource estimation model is available for SOA applications, the auto-scaler can provision resources in one shot to every service with minimum provision time. Models using queuing networks can be explored to fulfill the gap. It also calls for efficient online optimization algorithms to decide how each service should be provisioned in real-time to minimize cost.

Provision using Rebated Pricing Models

Besides Amazon's spot Cloud, providers like Google and Microsoft have introduced their rebated pricing models. However, researchers have only concentrated on exploring how to utilize Amazon's spot market while have been oblivious to other providers offerings. New works can aim to use cost models from other providers to provision resources. It is also interesting to research the use of rebated resources in a multiple Cloud environment with resources from multiple data centers of the same provider or multiple providers to minimize cost under QoS constraints. Besides, the proposed approach only co-utilizes on demand resources and rebated resources. Techniques that are provisioned with a mixture of on-demand, reserved, and rebated resources would be welcomed by industry and can be another potential future research direction.

Event-based Workload Prediction

Existing auto-scalers mostly rely on past workload history to predict future workload. With the growing popularity of social media and other real-time information channels, it is interesting to investigate the use of these sources of information to predict workload burst accurately. Although it is difficult to design a general-purpose predictor of this kind for various applications, there is potential to integrate the predictors catered for a particular type of applications into the auto-scalers, such as news applications whose workloads are boosted by events in the physical world, and outdoor applications whose workloads are subject to weather conditions.

Energy and Carbon-Aware Auto-scaling

The existing works only focus on financial cost and QoS aspects. As another primary concern of the ICT sector, energy and carbon footprint should also be considered in the auto-scaling systems. Nowadays, many data centers have been equipped with on-site generators utilizing renewable energy. However, these sources of energy, such as wind and solar, are unstable. Auto-scalers can preferentially provision resources in data centers that have renewable energy available to maximize use of on-site renewable energy. Within a single data center, auto-scalers can utilize vertical scaling as much as possible to avoid starting new physical machines to save energy.

Container-based Applications

The emergence of containers, especially the container-supported micro-services and service pods, has aroused a new revolution in web application resource management. However, dedicated auto-scaling solutions that cater for specific characteristics of the container era are still left to be explored. Though this thesis focuses on deployment based on VMs, we believe some notions and techniques mentioned in this thesis can inspire research of container-based auto-scalers since the core requirements of them are similar. However, they do are different in some aspects, e.g., containers are more flexible in sizes, and quicker and more lightweight to provision.

7.2.4 Overload Handling

For overload handling, we present the following research directions.

Capacity Plan for Over-Provisioning

By using our proposed approach in Chapter 6, application providers are required to over-provision some resources to prepare for the sudden shortages of resources. However, to over-provision how much and how to over-provision the resources are not explored. It is essential to have a global capacity planning algorithm to make these decisions to minimize cost overhead while our approach is functioning.

Integrated Overload Handling

The proposed approach uses geographical load balancing and admission control to handle resource overload. Other methods, such as brownout, which temporarily terminate some optional functions to ensure the execution of critical services, and approximation which reduces the precision of results to speed up processing when resources are not enough, can also be considered and integrated together to build a stronger overload handling technique. New policies and algorithms should be developed to make proper decisions using combinations of available overload handling methods to maximize QoS and minimize lost.

7.3 Final Remarks

Cloud computing has revealed a great potential for web application providers to save cost and grow fast in this ever competitive market. Besides, utilizing multiple Cloud data centers opens up more opportunities to solve the challenges in management and deployment of web applications. In this thesis, we explored problems that hinder application providers to effectively and cost-efficiently use Cloud resources in geographical dispersed data centers. The proposed approaches in this thesis are instrumental in

the building of next generation multi-Cloud middleware tools, which will enable cost-efficient, QoS-satisfied, and user-friendly management of web applications in Clouds.

Bibliography

[1] F. Al-Haidari, M. Sqalli, and K. Salah, "Impact of CPU utilization thresholds and scaling size on autoscaling cloud resources," in *Proceedings of 2013 IEEE 5th International Conference on Cloud Computing Technology and Science*, vol. 2, Dec 2013, pp. 256–261.

[2] H. M. Alabool and A. K. Mahmood, "Trust-based service selection in public cloud computing using fuzzy modified VIKOR method," *Australian Journal of Basic and Applied Sciences*, vol. 7, no. 9, pp. 211–220, 2013.

[3] alachisoft, "nCache," 2016. [Online]. Available: http://www.alachisoft.com/ncache/multi-site-dotnet-session-sharing.html

[4] A. Ali-Eldin, J. Tordsson, and E. Elmroth, "An adaptive hybrid elasticity controller for cloud infrastructures," in *Proceedings of 2012 IEEE Network Operations and Management Symposium (NOMS)*. IEEE, 2012, Conference Proceedings, pp. 204–212.

[5] A. Ali-Eldin, J. Tordsson, E. Elmroth, and M. Kihl, "Workload classification for efficient auto-scaling of cloud resources," Technical Report, 2005.[Online]. Available: http://www.cs.umu.se/research/uminf/reports/2013/013/part1.pdf, Tech. Rep., 2013.

[6] F. J. Almeida Morais, F. Vilar Brasileiro, R. Vigolvino Lopes, R. Araujo Santos, W. Satterfield, and L. Rosa, "Autoflex: Service agnostic auto-scaling framework for IaaS deployment models," in *Proceedings of 2013 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2013, Conference Proceedings, pp. 42–49.

[7] H. A. Alzoubi, S. Lee, M. Rabinovich, O. Spatscheck, and J. Van Der Merwe, "A practical architecture for an anycast CDN," *ACM Trans. Web*, vol. 5, no. 4, pp. 17:1–17:29, Oct. 2011.

[8] A. Amato, B. D. Martino, and S. Venticinque, "Cloud brokering as a service," in *Proceedings of 2013 Eighth International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC)*, Oct 2013, pp. 9–16.

[9] Amazon, "Amazon EC2 spot instances." [Online]. Available: http://aws.amazon.com/ec2/spot-instances/

[10] ——, "EC2 spot instance termination notices," 2015. [Online]. Available: https://aws.amazon.com/blogs/aws/new-ec2-spot-instance-termination-notices/

[11] ——, "Amazon auto scaling service," 2016. [Online]. Available: http://aws.amazon.com/autoscaling/

[12] ——, "Amazon spot fleet api," 2016. [Online]. Available: https://aws.amazon.com/blogs/aws/new-resource-oriented-bidding-for-ec2-spot-instances/

[13] ——, "Government & education case studies," 2016. [Online]. Available: https://aws.amazon.com/solutions/case-studies/government-education/

[14] ——, "Route 53," 2016. [Online]. Available: https://aws.amazon.com/route53/

[15] M. Andreolini, S. Casolari, and M. Colajanni, "Autonomic request management algorithms for geographically distributed internet-based systems," in *Proceedings of 2008 Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems*, Oct 2008, pp. 171–180.

[16] L. Aniello, S. Bonomi, F. Lombardi, A. Zelli, and R. Baldoni, *An Architecture for Automatic Scaling of Replicated Services*, ser. Lecture Notes in Computer Science. Springer International Publishing, 2014, book section 9, pp. 122–137.

[17] Apache, "Cassandra," 2015. [Online]. Available: http://cassandra.apache.org/

[18] D. Ardagna, S. Casolari, M. Colajanni, and B. Panicucci, "Dual time-scale distributed capacity allocation and load redirect algorithms for cloud systems," *Journal of Parallel and Distributed Computing*, vol. 72, no. 6, pp. 796 – 808, 2012.

[19] P. Bailis, S. Venkataraman, M. J. Franklin, J. M. Hellerstein, and I. Stoica, "Quantifying eventual consistency with PBS," *The VLDB Journal*, vol. 23, no. 2, pp. 279–302, 2014.

[20] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh, "Megastore: Providing scalable, highly available storage for interactive services," in *Proceedings of the Conference on Innovative Data system Research (CIDR)*, 2011, pp. 223–234.

[21] E. Barrett, E. Howley, and J. Duggan, "Applying reinforcement learning towards automating resource allocation and application scalability in the cloud," *Concurrency and Computation: Practice and Experience*, vol. 25, no. 12, pp. 1656–1674, 2013.

[22] A. Benoit, V. Rehn-Sonigo, and Y. Robert, "Replica placement and access policies in tree networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 19, no. 12, pp. 1614–1627, Dec 2008.

[23] J. Bi, Z. Zhu, R. Tian, and Q. Wang, "Dynamic provisioning modeling for virtualized multi-tier applications in cloud data center," in *Proceedings of 2010 IEEE 3rd International Conference on Cloud Computing*, July 2010, pp. 370–377.

[24] C. Binnig, A. Salama, E. Zamanian, M. El-Hindi, S. Feil, and T. Ziegler, "Spotgres - parallel data analytics on spot instances," in *Proceedings of 2015 31st IEEE International Conference on Data Engineering Workshops (ICDEW)*, April 2015, pp. 14–21.

[25] M. Bjrkqvist, L. Y. Chen, and W. Binder, "Cost-driven service provisioning in hybrid clouds," in *Proceedings of 2012 Fifth IEEE International Conference on Service-Oriented Computing and Applications (SOCA)*, Dec 2012, pp. 1–8.

[26] P. Bodk, R. Griffith, C. Sutton, A. Fox, M. Jordan, and D. Patterson, "Statistical machine learning makes automatic control practical for internet datacenters," in

Proceedings of the 2009 conference on Hot topics in cloud computing, 2009, Conference Proceedings, pp. 12–12.

[27] X. Bu, J. Rao, and C. Z. Xu, "Coordinated self-configuration of virtual machines and appliances using a model-free learning approach," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 4, pp. 681–690, April 2013.

[28] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic, "Cloud computing and emerging {IT} platforms: Vision, hype, and reality for delivering computing as the 5th utility," *Future Generation Computer Systems*, vol. 25, no. 6, pp. 599 – 616, 2009.

[29] N. M. Calcavecchia, B. A. Caprarescu, E. Di Nitto, D. J. Dubois, and D. Petcu, "DEPAS: a decentralized probabilistic algorithm for auto-scaling," *Computing*, vol. 94, no. 8, pp. 701–730, 2012.

[30] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. De Rose, and R. Buyya, "CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Software: Practice and Experience*, vol. 41, no. 1, pp. 23–50, 2011.

[31] V. Cardellini, M. Colajanni, and P. S. Yu, "Request redirection algorithms for distributed web systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, no. 4, pp. 355–368, April 2003.

[32] E. Caron, F. Desprez, and A. Muresan, "Pattern matching based forecast of non-periodic repetitive behavior for cloud clients," *Journal of Grid Computing*, vol. 9, no. 1, pp. 49–64, 2011.

[33] A. Chandra and P. Shenoy, "Effectiveness of dynamic resource allocation for handling internet flash crowds," *TR03-37, Department of Computer Science, University of Massachusetts, USA*, 2003.

[34] T. D. Chandra, R. Griesemer, and J. Redstone, "Paxos made live: An engineering perspective," in *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles*

of Distributed Computing, ser. PODC '07. New York, NY, USA: ACM, 2007, pp. 398–407.

[35] C. Chen, B. S. Lee, and X. Tang, "Improving Hadoop monetary efficiency in the cloud using spot instances," in *Proceedings of 2014 IEEE 6th International Conference on Cloud Computing Technology and Science (CloudCom)*, Dec 2014, pp. 312–319.

[36] C. Chen, S. Yan, G. Zhao, B. S. Lee, and S. Singhal, "A systematic framework enabling automatic conflict detection and explanation in cloud service selection for enterprises," in *Proceedings of 2012 IEEE 5th International Conference on Cloud Computing (CLOUD)*, June 2012, pp. 883–890.

[37] G. Chen, W. He, J. Liu, S. Nath, L. Rigas, L. Xiao, and F. Zhao, "Energy-aware server provisioning and load dispatching for connection-intensive internet services," in *Proceedings of 5th USENIX Symposium on Networked Systems Design and Implementation*, vol. 8, 2008, Conference Proceedings, pp. 337–350.

[38] X. Chen and J. Heidemann, "Flash crowd mitigation via adaptive admission control based on application-level observations," *ACM Transactions on Internet Technology*, vol. 5, no. 3, pp. 532–569, Aug. 2005.

[39] T. C. Chieu, A. Mohindra, and A. A. Karve, "Scalability and performance of web applications in a compute cloud," in *Proceedings of 2011 IEEE 8th International Conference on e-Business Engineering (ICEBE)*, 2011, Conference Proceedings, pp. 317–323.

[40] T. C. Chieu, A. Mohindra, A. A. Karve, and A. Segal, "Dynamic scaling of web applications in a virtualized cloud computing environment," in *Proceedings of IEEE International Conference on e-Business Engineering, 2009. (ICEBE '09)*, 2009, Conference Proceedings, pp. 281–286.

[41] E. Chlebus and J. Brazier, "Nonstationary poisson modeling of web browsing session arrivals," *Information Processing Letters*, vol. 102, no. 5, pp. 187 – 190, 2007.

[42] N. Chohan, C. Castillo, M. Spreitzer, M. Steinder, A. Tantawi, and C. Krintz, "See spot run: using spot instances for mapreduce workflows," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*. USENIX Association, pp. 7–7.

[43] H. Y. Chu and Y. Simmhan, "Cost-efficient and resilient job life-cycle management on hybrid clouds," in *Proceedings of 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, May 2014, pp. 327–336.

[44] Codership, "Certification-based commit," 2014. [Online]. Available: http://galeracluster.com/documentation-webpages/certificationbasedreplication.html

[45] ——, "Galera cluster," 2015. [Online]. Available: http://galeracluster.com/products/

[46] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, "Spanner: Google’s globally distributed database," *ACM Trans. Comput. Syst.*, vol. 31, no. 3, pp. 8:1–8:22, Aug. 2013.

[47] S. Costache, N. Parlavantzas, C. Morin, and S. Kortas, "Themis: Economy-based automatic resource scaling for cloud systems," in *Proceedings of 2012 IEEE 9th International Conference on High Performance Computing and Communication & 2012 IEEE 14th International Conference on Embedded Software and Systems (HPCC-ICESS)*, June 2012, pp. 367–374.

[48] C.S.M.I.C, "Service measurement index." [Online]. Available: http://csmic.org/understanding-smi/

[49] R. L. F. Cunha, M. D. Assuncao, C. Cardonha, and M. A. S. Netto, "Exploiting user patience for scaling resource capacity in cloud services," in *Proceedings of 2014 IEEE 7th International Conference on Cloud Computing*, June 2014, pp. 448–455.

[50] A. da Silva Dias, L. H. V. Nakamura, J. C. Estrella, R. H. C. Santana, and M. J. Santana, "Providing IaaS resources automatically through prediction and monitoring approaches," in *Proceedings of 2014 IEEE Symposium on Computers and Communication (ISCC)*, 2014, Conference Proceedings, pp. 1–7.

[51] A. V. Dastjerdi, S. G. H. Tabatabaei, and R. Buyya, "An effective architecture for automated appliance management system applying ontology-based cloud discovery," in *Proceedings of 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid)*, May 2010, pp. 104–112.

[52] W. Dawoud, I. Takouna, and C. Meinel, *Elastic Virtual Machine for Fine-Grained Cloud Resource Provisioning*, ser. Communications in Computer and Information Science. Springer Berlin Heidelberg, 2012, vol. 269, book section 2, pp. 11–25.

[53] U. de Paula Junior, L. M. A. Drummond, D. de Oliveira, Y. Frota, and V. C. Barbosa, "Handling flash-crowd events to improve the performance of web applications," in *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. New York, NY, USA: ACM, 2015, pp. 769–774.

[54] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," in *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6. ACM, 2007, pp. 205–220.

[55] R. P. Doyle, J. S. Chase, O. M. Asad, W. Jin, and A. Vahdat, "Model-based resource provisioning in a web service utility," in *Proceedings of the 2003 USENIX Symposium on Internet Technologies and Systems*, vol. 4, 2003, Conference Proceedings, pp. 5–5.

[56] X. Dutreilh, A. Moreau, J. Malenfant, N. Rivierre, and I. Truck, "From data center resource allocation to control theory and back," in *Proceedings of 2010 IEEE 3rd International Conference on Cloud Computing*, July 2010, pp. 410–417.

[57] X. Dutreilh, S. Kirgizov, O. Melekhova, J. Malenfant, N. Rivierre, and I. Truck, "Using reinforcement learning for autonomic resource allocation in clouds: towards a

fully automated workflow," in *Proceedings of 2011 International Conference on Autonomic and Autonomous Systems*, 2011, pp. 67–74.

[58] S. Dutta, S. Gera, V. Akshat, and B. Viswanathan, "SmartScale: Automatic application scaling in enterprise clouds," in *Proceedings of 2012 IEEE 5th International Conference on Cloud Computing (CLOUD)*, 2012, Conference Proceedings, pp. 221–228.

[59] eApps, "eapps," 2016. [Online]. Available: http://www.eapps.com/

[60] A. A. Eldin, A. Rezaie, A. Mehta, S. Razroev, S. S. d. Luna, O. Seleznjev, J. Tordsson, and E. Elmroth, "How will your workload look like in 6 years? analyzing Wikimedia's workload," in *Proceedings of 2014 IEEE International Conference on Cloud Engineering (IC2E)*, March 2014, pp. 349–354.

[61] M. Fallah, M. G. Arani, and M. Maeen, "NASLA: Novel auto scaling approach based on learning automata for web application in cloud computing environment," *International Journal of Computer Applications*, vol. 113, no. 2, 2015.

[62] W. Fang, Z. Lu, J. Wu, and Z. Cao, "RPPS: A novel resource prediction and provisioning scheme in cloud data center," in *Proceedings of 2012 IEEE Ninth International Conference on Services Computing (SCC)*, June 2012, pp. 609–616.

[63] H. Fernandez, G. Pierre, and T. Kielmann, "Autoscaling web applications in heterogeneous cloud infrastructures," in *Proceedings of 2014 IEEE International Conference on Cloud Engineering (IC2E)*, 2014, Conference Proceedings, pp. 195–204.

[64] A. J. Ferrer, F. Hernández, J. Tordsson, E. Elmroth, A. Ali-Eldin, C. Zsigri, R. Sirvent, J. Guitart, R. M. Badia, K. Djemame, W. Ziegler, T. Dimitrakos, S. K. Nair, G. Kousiouris, K. Konstanteli, T. Varvarigou, B. Hudzia, A. Kipp, S. Wesner, M. Corrales, N. Forgó, T. Sharif, and C. Sheridan, "OPTIMIS: A holistic approach to cloud service provisioning," *Future Generation Computer Systems*, vol. 28, no. 1, pp. 66 – 77, 2012.

[65] S. Frey, C. Luthje, C. Reich, and N. Clarke, "Cloud QoS scaling by fuzzy logic," in *Proceedings of 2014 IEEE International Conference on Cloud Engineering (IC2E)*, 2014, Conference Proceedings, pp. 343–348.

[66] A. Gambi, M. Pezze, and G. Toffetti, "Kriging-based self-adaptive cloud controllers," *IEEE Transactions on Services Computing*, vol. PP, no. 99, pp. 1–1, 2015.

[67] A. Gambi, G. Toffetti, C. Pautasso, and M. Pezz, "Kriging controllers for cloud applications," *IEEE Internet Computing*, vol. 17, no. 4, pp. 40–47, July 2013.

[68] A. Gandhi, P. Dube, A. Karve, A. Kochut, and L. Zhang, "Modeling the impact of workload on cloud resource scaling," in *Computer Architecture and High Performance Computing (SBAC-PAD), 2014 IEEE 26th International Symposium on*, Oct 2014, pp. 310–317.

[69] ——, "Adaptive, model-driven autoscaling for cloud applications," in *Proceedings of the 11th International Conference on Autonomic Computing (ICAC 14)*. Philadelphia, PA: USENIX Association, Jun. 2014, pp. 57–64.

[70] A. Gandhi, T. Zhu, M. Harchol-Balter, and M. A. Kozuch, *SOFTScale: Stealing Opportunistically for Transient Scaling*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 142–163.

[71] S. K. Garg, S. Versteeg, and R. Buyya, "A framework for ranking of cloud computing services," *Future Generation Computer Systems*, vol. 29, no. 4, pp. 1012 – 1023, 2013, special Section: Utility and Cloud Computing.

[72] I. Gergin, B. Simmons, and M. Litoiu, "A decentralized autonomic architecture for performance control in the cloud," in *Proceedings of 2014 IEEE International Conference on Cloud Engineering (IC2E)*, 2014, Conference Proceedings, pp. 574–579.

[73] H. Ghanbari, M. Litoiu, P. Pawluk, and C. Barna, "Replica placement in cloud through simple stochastic model predictive control," in *2014 IEEE 7th International Conference on Cloud Computing*, June 2014, pp. 80–87.

[74] H. Ghanbari, B. Simmons, M. Litoiu, C. Barna, and G. Iszlai, "Optimal autoscaling in a IaaS cloud," in *Proceedings of the 9th International Conference on Autonomic Computing*, ser. ICAC '12. New York, NY, USA: ACM, 2012, pp. 173–178.

[75] B. V. Gnedenko and I. N. Kovalenko, *Introduction to queueing theory*. Birkhauser Boston Inc., 1989.

[76] Z. Gong, X. Gu, and J. Wilkes, "PRESS: Predictive elastic resource scaling for cloud systems," in *Proceedings of 2010 International Conference on Network and Service Management*, Oct 2010, pp. 9–16.

[77] D. Grimaldi, V. Persico, A. Pescape, A. Salvi, and S. Santini, "A feedback-control approach for resource management in public clouds," in *2015 IEEE Global Communications Conference (GLOBECOM)*, Dec 2015, pp. 1–7.

[78] N. Grozev and R. Buyya, "Multi-cloud provisioning and load distribution for three-tier applications," *ACM Transactions on Autonomous and Adaptive Systems*, vol. 9, no. 3, pp. 13:1–13:21, 2014.

[79] ——, "Dynamic selection of virtual machines for application servers in cloud environments," *CoRR*, vol. abs/1602.02339, 2016.

[80] S. M. Habib, S. Ries, and M. Muhlhauser, "Towards a trust management system for cloud computing," in *Proceedings of 2011 IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications*, Nov 2011, pp. 933–939.

[81] S. M. Habib, V. Varadharajan, and M. Mühlhäuser, "A framework for evaluating trust of service providers in cloud marketplaces," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, ser. SAC '13. New York, NY, USA: ACM, 2013, pp. 1963–1965.

[82] R. Han, M. M. Ghanem, L. Guo, Y. Guo, and M. Osmond, "Enabling cost-aware and adaptive elasticity of multi-tier cloud applications," *Future Generation Computer Systems*, vol. 32, pp. 82–98, 2014.

[83] haproxy.org, "Haproxy 1.6 configuration manual." [Online]. Available: https://cbonte.github.io/haproxy-dconv/configuration-1.6.html

[84] C. Harmony, "List of major providers." [Online]. Available: https://cloudharmony.com/cloudsquare

[85] J. He, D. Wu, Y. Zeng, X. Hei, and Y. Wen, "Toward optimal deployment of cloud-assisted video distribution services," *Proceedings of IEEE Transactions on Circuits and Systems for Video Technology*, vol. 23, no. 10, pp. 1717–1728, Oct 2013.

[86] X. He, P. Shenoy, R. Sitaraman, and D. Irwin, "Cutting the cost of hosting online services using cloud spot markets," in *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2015, pp. 207–218.

[87] N. R. Herbst, N. Huber, S. Kounev, and E. Amrehn, "Self-adaptive workload classification and forecasting for proactive resource provisioning," *Concurrency and Computation: Practice and Experience*, vol. 26, no. 12, pp. 2053–2078, 2014.

[88] N. R. Herbst, S. Kounev, and R. Reussner, "Elasticity in cloud computing: What it is, and what it is not," in *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*. San Jose, CA: USENIX, 2013, pp. 23–27.

[89] N. Huber, F. Brosig, and S. Kounev, "Model-based self-adaptive resource allocation in virtualized environments," in *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, ser. SEAMS '11. New York, NY, USA: ACM, 2011, pp. 90–99.

[90] W. Iqbal, M. N. Dailey, and D. Carrera, "Unsupervised learning of dynamic resource provisioning policies for cloud-hosted multitier web applications," *IEEE Systems Journal*, vol. PP, no. 99, pp. 1–12, 2015.

[91] W. Iqbal, M. Dailey, and D. Carrera, *SLA-driven adaptive resource management for web applications on a heterogeneous compute cloud*. Springer, 2009, pp. 243–253.

[92] W. Iqbal, M. N. Dailey, D. Carrera, and P. Janecek, "Adaptive resource provisioning for read intensive multi-tier applications in the cloud," *Future Generation Computer Systems*, vol. 27, no. 6, pp. 871–879, 2011.

[93] S. Islam, J. Keung, K. Lee, and A. Liu, "An empirical study into adaptive resource provisioning in the cloud," in *Proceedings of IEEE International Conference on Utility and Cloud Computing (UCC 2010)*, 2010, Conference Proceedings, p. 8.

[94] ——, "Empirical prediction models for adaptive resource provisioning in the cloud," *Future Generation Computer Systems*, vol. 28, no. 1, pp. 155–162, 2012.

[95] P. Jamshidi, C. Pahl, and N. C. Mendona, "Managing uncertainty in autonomic cloud elasticity controllers," *IEEE Cloud Computing*, vol. 3, no. 3, pp. 50–60, May 2016.

[96] I. Jangjaimon and T. Nian-Feng, "Effective cost reduction for elastic clouds under spot instance pricing through adaptive checkpointing," *IEEE Transactions on Computers*, vol. 64, no. 2, pp. 396–409, 2015.

[97] B. Javadi, J. Abawajy, and R. Buyya, "Failure-aware resource provisioning for hybrid cloud infrastructure," *Journal of Parallel and Distributed Computing*, vol. 72, no. 10, pp. 1318–1331, 2012.

[98] D. Jiang, G. Pierre, and C.-H. Chi, "Autonomous resource provisioning for multi-service web applications," in *Proceedings of the 19th international conference on World wide web*. ACM, 2010, Conference Proceedings, pp. 471–480.

[99] ——, "Resource provisioning of web applications in heterogeneous clouds," in *Proceedings of the 2nd USENIX conference on Web application development*. USENIX Association, 2011, Conference Proceedings, pp. 5–5.

[100] J. Jiang, J. Lu, G. Zhang, and G. Long, "Optimal cloud resource auto-scaling for web applications," in *Proceedings of 2013 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE, 2013, pp. 58–65.

[101] L. Jiao, J. Li, T. Xu, W. Du, and X. Fu, "Optimizing cost for online social networks on geo-distributed clouds," *IEEE/ACM Transactions on Networking*, vol. 24, no. 1, pp. 99–112, Feb 2016.

[102] L. Jiao, J. Lit, W. Du, and X. Fu, "Multi-objective data placement for multi-cloud socially aware services," in *Proceedings of 2014 IEEE Conference on Computer Communications (IEEE INFOCOM)*, April 2014, pp. 28–36.

[103] X. Jing, Z. Ming, J. Fortes, R. Carpenter, and M. Yousif, "On the use of fuzzy modeling in virtualized data center management," in *Proceedings of Fourth International Conference on Autonomic Computing, 2007. (ICAC '07)*, 2007, Conference Proceedings, pp. 25–25.

[104] Y. Jingqi, L. Chuanchang, S. Yanlei, M. Zexiang, and C. Junliang, "Workload predicting-based automatic scaling in service clouds," in *Proceedings of 2013 IEEE Sixth International Conference on Cloud Computing (CLOUD)*, 2013, pp. 810–815.

[105] JOptimizer, "JOptimizer," 2016. [Online]. Available: http://www.joptimizer.com/

[106] A. Jøsang, R. Ismail, and C. Boyd, "A survey of trust and reputation systems for online service provision," *Decision Support Systems*, vol. 43, no. 2, pp. 618 – 644, 2007, emerging Issues in Collaborative Commerce.

[107] D. Jung, S. Chin, K. Chung, H. Yu, and J. Gil, *An Efficient Checkpointing Scheme Using Price History of Spot Instances in Cloud Computing Environment*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, vol. 6985, book section 16, pp. 185–200.

[108] G. Jung, K. R. Joshi, M. A. Hiltunen, R. D. Schlichting, and C. Pu, "Generating adaptation policies for multi-tier applications in consolidated server environments," in *Proceedings of 2008 International Conference on Autonomic Computing (ICAC '08)*, 2008, Conference Proceedings, pp. 23–32.

[109] E. Kalyvianaki, T. Charalambous, and S. Hand, "Self-adaptive and self-configured CPU resource provisioning for virtualized servers using kalman filters," in *Proceed-*

ings of the 6th International Conference on Autonomic Computing, ser. ICAC '09. New York, NY, USA: ACM, 2009, pp. 117–126.

[110] A. Kamra, V. Misra, and E. M. Nahum, "Yaksha: a self-tuning controller for managing the performance of 3-tiered web sites," in *Quality of Service, 2004. IWQOS 2004. Twelfth IEEE International Workshop on*, 2004, Conference Proceedings, pp. 47–56.

[111] J. Kang and K. M. Sim, *Cloudle: An Ontology-Enhanced Cloud Service Search Engine*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 416–427.

[112] Y. Kang, Z. Zheng, and M. R. Lyu, "A latency-aware co-deployment mechanism for cloud-based services," in *Proceedings of 2012 IEEE 5th International Conference on Cloud Computing (CLOUD)*, June 2012, pp. 630–637.

[113] Y. Kang, Y. Zhou, Z. Zheng, and M. R. Lyu, "A user experience-based cloud service redeployment mechanism," in *Proceedings of 2011 IEEE International Conference on Cloud Computing (CLOUD)*, July 2011, pp. 227–234.

[114] P. D. Kaur and I. Chana, "A resource elasticity framework for QoS-aware execution of cloud applications," *Future Generation Computer Systems*, vol. 37, no. 0, pp. 14–25, 2014.

[115] B. Kemme and G. Alonso, "Don't be lazy, be consistent: Postgres-R, a new way to implement database replication," in *Proceedings of the 26th International Conference on Very Large Data Bases*, ser. VLDB '00. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000, pp. 134–143.

[116] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, Jan 2003.

[117] M. Kiran, M. Jiang, D. J. Armstrong, and K. Djemame, "Towards a service lifecycle based methodology for risk assessment in cloud computing," in *Proceedings of 2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing (DASC)*, Dec 2011, pp. 449–456.

[118] C. Klein *et al.*, "Brownout: Building more robust cloud applications," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 700–711.

[119] Y. Koh, R. Knauerhase, P. Brett, M. Bowman, Z. Wen, and C. Pu, "An analysis of performance interference effects in virtual environments," in *Proceedings of 2007 IEEE International Symposium on Performance Analysis of Systems Software*, April 2007, pp. 200–209.

[120] KPMG, "2014 cloud survey report," 2014.

[121] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete, "MDCC: Multi-data center consistency," in *Proceedings of the 8th ACM European Conference on Computer Systems*, ser. EuroSys '13. New York, NY, USA: ACM, 2013, pp. 113–126.

[122] P. Lama and X. Zhou, "Autonomic provisioning with self-adaptive neural fuzzy control for end-to-end delay guarantee," in *2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, Aug 2010, pp. 151–160.

[123] ——, "Efficient server provisioning with end-to-end delay guarantee on multi-tier clusters," in *Proceedings of 17th International Workshop on Quality of Service (IWQoS 2009)*, 2009, Conference Proceedings, pp. 1–9.

[124] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, no. 2, pp. 133–169, May 1998.

[125] E. Lassettre, D. W. Coleman, Y. Diao, S. Froehlich, J. L. Hellerstein, L. Hsiung, T. Mummert, M. Raghavachari, G. Parker, L. Russell, M. Surendra, V. Tseng, N. Wadia, and P. Ye, *Dynamic Surge Protection: An Approach to Handling Unexpected Workload Surges with Resource Actions that Have Lead Times*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 82–92.

[126] B. Li, M. J. Golin, G. F. Italiano, X. Deng, and K. Sohraby, "On the optimal placement of web proxies in the internet," in *Proceedings of INFOCOM '99. Eighteenth Annual*

Joint Conference of the IEEE Computer and Communications Societies, vol. 3, Mar 1999, pp. 1282–1290 vol.3.

[127] H. Li and S. Venugopal, "Using reinforcement learning for controlling an elastic web application hosting platform," in *Proceedings of the 8th ACM International Conference on Autonomic Computing*, ser. ICAC '11. New York, NY, USA: ACM, 2011, pp. 205–208.

[128] H. C. Lim, S. Babu, and J. S. Chase, "Automated control for elastic storage," in *Proceedings of the 7th International Conference on Autonomic Computing*, ser. ICAC '10. New York, NY, USA: ACM, 2010, pp. 1–10.

[129] H. C. Lim, S. Babu, J. S. Chase, and S. S. Parekh, "Automated control in cloud computing: Challenges and opportunities," in *Proceedings of the 1st Workshop on Automated Control for Datacenters and Clouds*, ser. ACDC '09. New York, NY, USA: ACM, 2009, pp. 13–18.

[130] L. Liu, X. Yao, L. Qin, and M. Zhang, "Ontology-based service matching in cloud computing," in *Proceedings of 2014 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)*, July 2014, pp. 2544–2550.

[131] Z. Liu, M. Lin, A. Wierman, S. Low, and L. L. H. Andrew, "Greening geographical load balancing," *IEEE/ACM Trans. Netw.*, vol. 23, no. 2, pp. 657–671, Apr. 2015.

[132] J. Loff and J. Garcia, "Vadara: Predictive elasticity for cloud applications," in *Proceedings of 2014 IEEE 6th International Conference on Cloud Computing Technology and Science (CloudCom)*, 2014, Conference Proceedings, pp. 541–546.

[133] T. Lorido-Botran, J. Miguel-Alonso, and J. Lozano, "A review of auto-scaling techniques for elastic applications in cloud environments," *Journal of Grid Computing*, vol. 12, no. 4, pp. 559–592, 2014.

[134] S. Lu, X. Li, L. Wang, H. Kasim, H. Palit, T. Hung, E. F. T. Legara, and G. Lee, "A dynamic hybrid resource provisioning approach for running large-scale computational applications on cloud spot and on-demand instances," in *Proceedings of 2013*

International Conference on Parallel and Distributed Systems (ICPADS), Dec 2013, pp. 657–662.

[135] J. Luna Garcia, R. Langenberg, and N. Suri, "Benchmarking cloud security level agreements using quantitative policy trees," in *Proceedings of the 2012 ACM Workshop on Cloud Computing Security Workshop*, ser. CCSW '12. New York, NY, USA: ACM, 2012, pp. 103–112.

[136] H. Mahmoud, F. Nawab, A. Pucher, D. Agrawal, and A. El Abbadi, "Low-latency multi-datacenter databases using replicated commit," *Proc. VLDB Endow.*, vol. 6, no. 9, pp. 661–672, Jul. 2013.

[137] S. J. Malkowski, M. Hedwig, J. Li, C. Pu, and D. Neumann, "Automated control for elastic N-tier workloads based on empirical modeling," in *Proceedings of the 8th ACM International Conference on Autonomic Computing*, ser. ICAC '11. New York, NY, USA: ACM, 2011, pp. 131–140.

[138] M. Mao and M. Humphrey, "A performance study on the vm startup time in the cloud," in *Proceedings of 2012 IEEE 5th International Conference on Cloud Computing (CLOUD)*, June 2012, pp. 423–430.

[139] M. Mazzucco and M. Dumas, "Achieving performance and availability guarantees with spot instances," in *Proceedings of 2011 IEEE 13th International Conference on High Performance Computing and Communications (HPCC)*, Sept 2011, pp. 296–303.

[140] MediaWiki, "MediaWiki." [Online]. Available: https://www.mediawiki.org

[141] P. Mell and T. Grance, "The NIST definition of cloud computing," 2011.

[142] M. Menzel and R. Ranjan, "CloudGenius: Decision support for web server cloud migration," in *Proceedings of the 21st International Conference on World Wide Web*, ser. WWW '12. New York, NY, USA: ACM, 2012, pp. 979–988.

[143] A. Nadjaran Toosi and R. Buyya, "A fuzzy logic-based controller for cost and energy efficient load balancing in geo-distributed data centers," in *Proceedings of 8th*

IEEE/ACM International Conferencce on Utility and Cloud Computing (UCC). IEEE, 2015.

[144] S. Nepal, W. Sherchan, J. Hunklinger, and A. Bouguettaya, "A fuzzy trust management framework for service web," in *Proceedings of 2010 IEEE International Conference on Web Services (ICWS)*, July 2010, pp. 321–328.

[145] M. A. S. Netto, C. Cardonha, R. L. F. Cunha, and M. D. Assuncao, "Evaluating auto-scaling strategies for cloud computing environments," in *Proceedings of 2014 IEEE 22nd International Symposium on Modelling, Analysis Simulation of Computer and Telecommunication Systems*, Sept 2014, pp. 187–196.

[146] H. Nguyen, Z. Shen, X. Gu, S. Subbiah, and J. Wilkes, "Agile: Elastic distributed resource scaling for infrastructure-as-a-service," in *Proceedings of the USENIX International Conference on Automated Computing (ICAC13)*, San Jose, CA, 2013, Conference Proceedings.

[147] A. Y. Nikravesh, S. A. Ajila, and C. H. Lung, "Towards an autonomic auto-scaling prediction system for cloud resource provisioning," in *Proceedings of 2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, May 2015, pp. 35–45.

[148] A. Nisar, W. Iqbal, F. S. Bokhari, and F. Bukhari, "Hybrid auto-scaling of multi-tier web applications: A case of using amazon public cloud."

[149] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani, "Scaling memcache at Facebook," in *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. Lombard, IL: USENIX, 2013, pp. 385–398.

[150] T. H. Noor and Q. Z. Sheng, *Trust as a Service: A Framework for Trust Management in Cloud Environments*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 314–321.

[151] Ookla, "NetMetrics," 2016. [Online]. Available: https://www.ookla.com/netmetrics

[152] P. Padala, K.-Y. Hou, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, and A. Merchant, "Automated control of multiple virtualized resources," in *Proceedings of the 4th ACM European Conference on Computer Systems*, ser. EuroSys '09. New York, NY, USA: ACM, 2009, pp. 13–26.

[153] T. Patikirikorala, A. Colman, J. Han, and L. Wang, "A multi-model framework to implement self-managing control systems for QoS management," in *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. ACM, 2011, Conference Proceedings, pp. 218–227.

[154] F. Pedone, "The database state machine and group communication issues," Ph.D. dissertation, IC, Lausanne, 1999.

[155] D. Poola, K. Ramamohanarao, and R. Buyya, "Fault-tolerant workflow scheduling using spot instances on clouds," *Procedia Computer Science*, vol. 29, pp. 523–533, 2014.

[156] R. Prodan and V. Nae, "Prediction-based real-time resource provisioning for massively multiplayer online games," *Future Generation Computer Systems*, vol. 25, no. 7, pp. 785–793, 2009.

[157] L. Qiu, V. N. Padmanabhan, and G. M. Voelker, "On the placement of web server replicas," in *Proceedings of INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies*, vol. 3, 2001, pp. 1587–1596 vol.3.

[158] C. Qu, R. N. Calheiros, and R. Buyya, "A reliable and cost-efficient auto-scaling system for web applications using heterogeneous spot instances," *Journal of Network and Computer Applications*, 2016.

[159] S. Ranjan, R. Karrer, and E. Knightly, "Wide area redirection of dynamic content by internet data centers," in *IEEE INFOCOM 2004*, vol. 2, March 2004, pp. 816–826 vol.2.

[160] C. Redl, I. Breskovic, I. Brandic, and S. Dustdar, "Automatic SLA matching and provider selection in grid and cloud computing markets," in *Proceedings of 2012 ACM/IEEE 13th International Conference on Grid Computing*, Sept 2012, pp. 85–94.

[161] S. Ries, S. M. Habib, M. Mühlhäuser, and V. Varadharajan, *CertainLogic: A Logic for Modeling Trust and Uncertainty*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 254–261.

[162] RightScale, "Understanding the voting process," 2016. [Online]. Available: https://support.rightscale.com/12-Guides/RightScale_101/System_Architecture/RightScale_Alert_System/Alerts_based_on_Voting_Tags/Understanding_the_Voting_Process/

[163] G. Rodolakis, S. Siachalou, and L. Georgiadis, "Replicated server placement with QoS constraints," *IEEE Transactions on Parallel and Distributed Systems*, vol. 17, no. 10, pp. 1151–1162, Oct 2006.

[164] N. Roy, A. Dubey, and A. Gokhale, "Efficient autoscaling in the cloud using predictive models for workload forecasting," in *Proceedings of 2011 IEEE International Conference on Cloud Computing (CLOUD)*. IEEE, 2011, pp. 500–507.

[165] H. Rui, G. Li, M. M. Ghanem, and G. Yike, "Lightweight resource scaling for cloud applications," in *Proceedings of 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2012, Conference Proceedings, pp. 644–651.

[166] K. Salah, K. Elbadawi, and R. Boutaba, "An analytical model for estimating cloud resources of elastic services," *Journal of Network and Systems Management*, pp. 1–24, 2015.

[167] Y. Sangho, A. Andrzejak, and D. Kondo, "Monetary cost-aware checkpointing and migration on Amazon cloud spot instances," *IEEE Transactions on Services Computing*, vol. 5, no. 4, pp. 512–524, 2012.

[168] P. Saripalli and B. Walters, "QUIRC: A quantitative impact and risk assessment framework for cloud security," in *Proceedings of 2010 IEEE 3rd International Conference on Cloud Computing*, July 2010, pp. 280–288.

[169] J. Schad, J. Dittrich, and J.-A. Quian-Ruiz, "Runtime measurements in the cloud: observing, analyzing, and reducing variance," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 460–471, 2010.

[170] M. Sedaghat, F. Hernandez-Rodriguez, and E. Elmroth, "A virtual machine re-packing approach to the horizontal vs. vertical elasticity trade-off for cloud autoscaling," in *Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference*, ser. CAC '13. New York, NY, USA: ACM, 2013, pp. 6:1–6:10.

[171] P. N. Shankaranarayanan, A. Sivakumar, S. Rao, and M. Tawarmalani, "Performance sensitive replication in geo-distributed cloud datastores," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, June 2014, pp. 240–251.

[172] P. Sharma, S. Lee, T. Guo, D. Irwin, and P. Shenoy, "SpotCheck: Designing a derivative IaaS cloud on the spot market," in *Proceedings of the Tenth European Conference on Computer Systems*. ACM, 2015, pp. 16:1–16:15.

[173] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes, "Cloudscale: elastic resource scaling for multi-tenant cloud systems," in *Proceedings of the 2nd ACM Symposium on Cloud Computing*. ACM, 2011, Conference Proceedings, p. 5.

[174] D. B. Shmoys, E. Tardos, and K. Aardal, "Approximation algorithms for facility location problems," in *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, ser. STOC '97. New York, NY, USA: ACM, 1997, pp. 265–274.

[175] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Littlefield, D. Menestrina, S. Ellner, J. Cieslewicz, I. Rae, T. Stancescu, and H. Apte, "F1: A distributed sql database that scales," *Proc. VLDB Endow.*, vol. 6, no. 11, pp. 1068–1079, Aug. 2013.

[176] R. Singh, P. Sharma, D. Irwin, P. Shenoy, and K. K. Ramakrishnan, "Here today, gone tomorrow: Exploiting transient servers in datacenters," *IEEE Internet Computing*, vol. 18, no. 4, pp. 22–29, 2014.

[177] R. Singh, U. Sharma, E. Cecchet, and P. Shenoy, "Autonomic mix-aware provisioning for non-stationary data center workloads," in *Proceedings of the 7th International Conference on Autonomic Computing*, ser. ICAC '10. New York, NY, USA: ACM, 2010, pp. 21–30.

[178] M. Smit, P. Pawluk, B. Simmons, and M. Litoiu, "A web service for cloud metadata," in *Proceedings of 2012 IEEE Eighth World Congress on Services*, June 2012, pp. 361–368.

[179] S. Song, K. Hwang, R. Zhou, and Y. K. Kwok, "Trusted P2P transactions with fuzzy reputation aggregation," *IEEE Internet Computing*, vol. 9, no. 6, pp. 24–34, Nov 2005.

[180] S. Song, K. Hwang, and M. Macwan, *Fuzzy Trust Integration for Security Enforcement in Grid Computing*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 9–21.

[181] S. Spinner, S. Kounev, X. Zhu, L. Lu, M. Uysal, A. Holler, and R. Griffith, "Runtime vertical scaling of virtualized applications via online model estimation," in *2014 IEEE Eighth International Conference on Self-Adaptive and Self-Organizing Systems*, Sept 2014, pp. 157–166.

[182] S. N. Srirama and A. Ostovar, "Optimal resource provisioning for scaling enterprise applications on the cloud," in *Cloud Computing Technology and Science (CloudCom), 2014 IEEE 6th International Conference on*, Dec 2014, pp. 262–271.

[183] S. Subramanya, T. Guo, P. Sharma, D. Irwin, and P. Shenoy, "SpotOn: A batch computing service for the spot market," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*. ACM, 2015, pp. 329–341.

[184] M. Sun, T. Zang, X. Xu, and R. Wang, "Consumer-centered cloud services selection using ahp," in *Proceedings of 2013 International Conference on Service Sciences (ICSS)*, April 2013, pp. 1–6.

[185] S. Sundareswaran, A. Squicciarini, and D. Lin, "A brokerage-based approach for cloud service selection," in *Proceedings of 2012 IEEE 5th International Conference on Cloud Computing (CLOUD)*, June 2012, pp. 558–565.

[186] M. Supriya *et al.*, "Estimating trust value for cloud service providers using fuzzy logic," *International Journal of Computer Applications*, vol. 48, no. 19, pp. 28–34, 2012.

[187] D. N. B. Ta, T. Nguyen, S. Zhou, X. Tang, W. Cai, and R. Ayani, "Interactivity-constrained server provisioning in large-scale distributed virtual environments," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 2, pp. 304–312, Feb 2012.

[188] D. N. B. Ta and S. Zhou, "Server placement for enhancing the interactivity of large-scale distributed virtual environments," in *Proceedings of 2006 International Conference on Cyberworlds*, Nov 2006, pp. 123–132.

[189] A. Tahamtan, S. A. Beheshti, A. Anjomshoaa, and A. M. Tjoa, "A cloud repository and discovery framework based on a unified business and cloud service ontology," in *Proceedings of 2012 IEEE Eighth World Congress on Services*, June 2012, pp. 203–210.

[190] X. Tang, H. Chi, and S. T. Chanson, "Optimal replica placement under ttl-based consistency," *IEEE Transactions on Parallel and Distributed Systems*, vol. 18, no. 3, pp. 351–363, March 2007.

[191] G. Tesauro, "Online resource allocation using decompositional reinforcement learning," in *Proceedings of AAAI*, vol. 5, 2005, Conference Proceedings, pp. 886–891.

[192] G. Tesauro, N. K. Jong, R. Das, and M. N. Bennani, "On the use of hybrid reinforcement learning for autonomic resource allocation," *Cluster Computing*, vol. 10, no. 3, pp. 287–299, 2007.

[193] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi, "Calvin: Fast distributed transactions for partitioned database systems," in *Proceedings of the*

2012 ACM SIGMOD International Conference on Management of Data, ser. SIGMOD '12. New York, NY, USA: ACM, 2012, pp. 1–12.

[194] V. Torra, "A review of the construction of hierarchical fuzzy systems," *International Journal of Intelligent Systems*, vol. 17, no. 5, pp. 531–543, 2002.

[195] Transaction Processing Performance Council, "TPC-W Workload," 2015. [Online]. Available: http://www.tpc.org/tpcw/

[196] Twissandra, "Twissandra." [Online]. Available: https://github.com/twissandra/twissandra

[197] Z. u. Rehman, F. K. Hussain, and O. K. Hussain, "Towards multi-criteria cloud service selection," in *Proceedings of 2011 Fifth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS)*, June 2011, pp. 44–48.

[198] Z. u. Rehman, O. K. Hussain, and F. K. Hussain, "Iaas cloud selection using MCDM methods," in *Proceedings of 2012 IEEE Ninth International Conference on e-Business Engineering (ICEBE)*, Sept 2012, pp. 246–251.

[199] ——, "Multi-criteria IaaS service selection based on QoS history," in *Proceedings of 2013 IEEE 27th International Conference on Advanced Information Networking and Applications (AINA)*, March 2013, pp. 1129–1135.

[200] S. Upendra, P. Shenoy, S. Sahu, and A. Shaikh, "A cost-aware elasticity provisioning system for the cloud," in *Proceedings of 2011 31st International Conference on Distributed Computing Systems (ICDCS)*, 2011, Conference Proceedings, pp. 559–570.

[201] G. Urdaneta, G. Pierre, and M. van Steen, "Wikipedia workload analysis for decentralized hosting," *Elsevier Computer Networks*, vol. 53, no. 11, pp. 1830–1845, July 2009.

[202] B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal, and T. Wood, "Agile dynamic provisioning of multi-tier internet applications," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 3, no. 1, p. 1, 2008.

[203] E.-J. van Baaren, "Wikibench: A distributed, wikipedia based web application benchmark," *Master's thesis, VU University Amsterdam*, 2009.

[204] ——, "Wikipedia access trace," 2015. [Online]. Available: http://www.wikibench.eu/?page_id=60

[205] N. Vasić, D. Novaković, S. Miučin, D. Kostić, and R. Bianchini, "DejaVu: Accelerating resource allocation in virtualized environments," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVII. New York, NY, USA: ACM, 2012, pp. 423–436.

[206] D. Villela, P. Pradhan, and D. Rubenstein, "Provisioning servers in the application tier for e-commerce systems," *ACM Trans. Internet Technol.*, vol. 7, no. 1, Feb. 2007.

[207] W. Voorsluys and R. Buyya, "Reliable provisioning of spot instances for compute-intensive applications," in *Proceedings of 2012 IEEE 26th International Conference on Advanced Information Networking and Applications*, March 2012, pp. 542–549.

[208] P. Wang, "QoS-aware web services selection with intuitionistic fuzzy set under consumers vague perception," *Expert Systems with Applications*, vol. 36, no. 3, Part 1, pp. 4460 – 4466, 2009.

[209] S. Wang, Z. Liu, Q. Sun, H. Zou, and F. Yang, "Towards an accurate evaluation of quality of cloud service in service-oriented cloud computing," *Journal of Intelligent Manufacturing*, vol. 25, no. 2, pp. 283–291, 2014.

[210] B. Wilder, *Cloud architecture patterns: using microsoft azure*. " O'Reilly Media, Inc.", 2012, ch. Horizontally Scaling Compute Pattern.

[211] ——, *Cloud architecture patterns: using microsoft azure*. " O'Reilly Media, Inc.", 2012.

[212] Y. Wu, C. Wu, B. Li, L. Zhang, Z. Li, and F. C. M. Lau, "Scaling social media applications into geo-distributed clouds," in *Proceedings of 2012 IEEE INFOCOM*, March 2012, pp. 684–692.

[213] Z. Wu, M. Butkiewicz, D. Perkins, E. Katz-Bassett, and H. V. Madhyastha, "SPANStore: Cost-effective geo-replicated storage spanning multiple cloud services," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: ACM, 2013, pp. 292–308.

[214] J. Yang, C. Liu, Y. Shang, B. Cheng, Z. Mao, C. Liu, L. Niu, and J. Chen, "A cost-aware auto-scaling approach using the workload prediction in service clouds," *Information Systems Frontiers*, vol. 16, no. 1, pp. 7–18, 2014.

[215] R. Yanggratoke, J. Ahmed, J. Ardelius, C. Flinta, A. Johnsson, D. Gillblad, and R. Stadler, "Predicting service metrics for cluster-based services using real-time analytics," in *Proceedings of 2015 11th International Conference on Network and Service Management (CNSM)*, Nov 2015, pp. 135–143.

[216] L. Yazdanov and C. Fetzer, "VScaler: Autonomic virtual machine scaling," in *Proceedings of 2013 IEEE Sixth International Conference on Cloud Computing (CLOUD)*, 2013, Conference Proceedings, pp. 212–219.

[217] ——, "Vertical scaling for prioritized vms provisioning," in *Proceedings of 2012 Second International Conference on Cloud and Green Computing (CGC)*. IEEE, 2012, Conference Proceedings, pp. 118–125.

[218] N. Yipei, L. Bin, L. Fangming, L. Jiangchuan, and L. Bo, "When hybrid cloud meets flash crowd: Towards cost-effective service provisioning," in *Proceedings of 2015 IEEE Conference on Computer Communications (INFOCOM)*, 2015, pp. 1044–1052.

[219] Y. You, G. Huang, J. Cao, E. Chen, J. He, Y. Zhang, and L. Hu, *Web Information Systems Engineering – WISE 2013: 14th International Conference, Nanjing, China, October 13-15, 2013, Proceedings, Part II*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, ch. GEAM: A General and Event-Related Aspects Model for Twitter Event Detection, pp. 319–332.

[220] M. Zafer, Y. Song, and K. W. Lee, "Optimal bids for spot vms in a cloud for deadline constrained jobs," in *Proceedings of 2012 IEEE 5th International Conference on Cloud Computing (CLOUD)*, June 2012, pp. 75–82.

[221] W. Zeng, Y. Zhao, and J. Zeng, "Cloud service and service selection algorithm research," in *Proceedings of the First ACM/SIGEVO Summit on Genetic and Evolutionary Computation*, ser. GEC '09. New York, NY, USA: ACM, 2009, pp. 1045–1048.

[222] H. Zhang, G. Jiang, K. Yoshihira, H. Chen, and A. Saxena, "Intelligent workload factoring for a hybrid cloud computing model," in *Proceedings of 2009 World Conference on Services*, 2009, pp. 701–708.

[223] Q. Zhang, L. Cherkasova, and E. Smirni, "A regression-based analytic model for dynamic resource provisioning of multi-tier applications," in *Proceedings of Fourth International Conference on Autonomic Computing (ICAC'07)*, June 2007, pp. 27–27.

[224] Q. Zhang, Q. Zhu, M. F. Zhani, R. Boutaba, and J. L. Hellerstein, "Dynamic service placement in geographically distributed clouds," *IEEE Journal on Selected Areas in Communications*, vol. 31, no. 12, pp. 762–772, December 2013.

[225] Y. Zhang, Y. Wang, and X. Wang, "GreenWare: Greening cloud-scale data centers to maximize the use of renewable energy," in *Proceedings of ACM/IFIP/USENIX 12th International Middleware Conference*. Springer, 2011, pp. 143–164.

[226] H. Zhao, M. Pan, X. Liu, X. Li, and Y. Fang, "Optimal resource rental planning for elastic applications in cloud market," in *Proceedings of 2012 IEEE 26th International Parallel Distributed Processing Symposium (IPDPS)*, May 2012, pp. 808–819.

[227] J. Zhu, Z. Zheng, Y. Zhou, and M. R. Lyu, "Scaling service-oriented applications into geo-distributed clouds," in *Proceedings of 2013 IEEE 7th International Symposium on Service Oriented System Engineering (SOSE)*, March 2013, pp. 335–340.

[228] Q. Zhu and G. Agrawal, "Resource provisioning with budget constraints for adaptive applications in cloud environments," *IEEE Transactions on Services Computing*, vol. 5, no. 4, pp. 497–511, 2012.