Engineering Self-Adaptive Applications on Software Defined Infrastructure

Nasim Beigi-Mohammadi

A dissertation submitted to the Faculty of Graduate Studies in partial fulfilment of the requirements for the degree of Doctorate of Philosophy

Graduate Program in Computer Science
York University
Toronto, Ontario, Canada

June 2019

©Nasim Beigi-Mohammadi, 2019

Abstract

Cloud computing is a flexible platform that offers faster innovation, elastic resources, and economies of scale. However, it is challenging to ensure non-functional properties such as performance, cost and security of applications hosted in cloud. Applications should be adaptive to the fluctuating workload to meet the desired performance goals, in one hand, and on the other, operate in an economic manner to reduce the operational cost. Moreover, cloud applications are attractive target of security threats such as distributed denial of service attacks that target the availability of applications and increase the cost. Given such circumstances, it is vital to engineer applications that are able to self-adapt to such volatile conditions. In this thesis, we investigate techniques and mechanisms to engineer model-based application autonomic management systems that strive to meet performance, cost and security objectives of software systems running in cloud. In addition to using the elasticity feature of cloud, our proposed autonomic management systems employ run-time network adaptations using the emerging software defined networking and network function virtualization. We propose a novel approach to self-protecting applications where the application traffic is dynamically managed between public and private cloud depending on the condition of the traffic. Our management approach is able to adapt the bandwidth rates of application traffic to meet performance and cost objectives. Through run-time performance models as well as optimization, the management system maximizes the profit each time the application requires to adapt. Our autonomous management solutions are implemented and evaluated analytically as well as on multiple public and community clouds to demonstrate their applicability and effectiveness in real world environment.

To my lovely husband, Hamzeh, and my dear son, Dastan.

Acknowledgments

I would like to express my sincere gratitude to my advisor Prof. Marin Litoiu, for the continuous encouragement of my Ph.D study, for his patience, motivation, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor for my PhD study.

I would also like to thank Dr. Mark Shtern that has been a tremendous mentor for me. Mark's friendly guidance and expert advice have been invaluable throughout all stages of the work.

I am also thankful to my lab mate, Dr. Cornel Barna, who was kind enough to share his PhD experience with me and guided me to use his framework, Hogna.

I should also thank Dr. Uyen Trang Nguyen, who provided me with precious suggestions and guidance, which contributed to the success of the scholarships and awards that I received during my PhD study.

I am grateful to Natural Sciences and Engineering Research Council of Canada (NSERC) for generously supporting my PhD stuy through CGS-D scholarship.

Last but not least, I wish to thank my loving and supportive husband, Dr. Hamzeh Khazaei, who always provides me unending inspiration and encouragement.

Contents

Abstract **ii**

Dedication **iii**

Acknowledgements **iv**

Contents **v**

List of Tables **ix**

List of Figures **xi**

Co-Authorship **xv**

1 Introduction **1**

2 Background and Related Work **8**

2.1 Background 8

2.1.1 Adaptive Systems 8

2.1.2 Software Defined Networking (SDN) and Network Function Virtualization (NFV) 13

2.1.3 Queuing Networks 16
2.1.4 Decision Tree Learning Models 19
2.2 Related Work 22
2.2.1 SDN: Application Awareness and Performance 22
2.2.2 DDoS Mitigation 25
2.2.3 Auto-scaling 27
2.3 Summary 29

3 Self-protecting Applications on Software Defined Infrastructure 30
3.1 CAAMP 32
3.1.1 Implementation 37
3.1.2 Experiment 41
3.2 A DevOps Framework for Quality-Driven Self-Protection in Web Software Systems 50
3.2.1 Framework Architecture and Components 51
3.2.2 DevOps Integration: A Quality-Driven Workflow 54
3.2.3 Experiment 56
3.2.4 Limitations and Assumptions 60
3.3 Summary 61

4 Self-managing Applications with Search-based Network Adaptation 63
4.1 Search-based Network Adaptation 66
4.1.1 Autonomic Manager 68
4.2 Implementation 71
4.3 Experiment 75

4.3.1 Event 1 81
4.3.2 Event 2 82
4.3.3 Event 3 83
4.4 Summary 84

5 Self-managing Applications with Machine-learning Based Network and Compute Adaptations **85**
5.1 A Machine-learning-based Autonomic Manager with Fine Granular Bandwidth Control 87
5.1.1 Model 88
5.2 Implementation 91
5.3 Experiment 93
5.3.1 Experiment 1: Hill-climbing Heuristic 93
5.3.2 Experiment 2: Model-based Adaptation 97
5.3.3 Analysis of the Results 99
5.4 Summary 100

6 Self-optimizing Applications with Scalable Machine-learning based Network and Compute Adaptations **103**
6.1 Adaptive Load Management of Web Applications on Software Defined Infrastructure 106
6.1.1 Application 107
6.2 Models and Run-time Adaptation 109
6.2.1 Performance Models 109
6.2.2 Optimization Module 112

6.3 Implementation 117
6.4 Experiment 118
6.4.1 Auto-scaling-only Approach 120
6.4.2 Model-based Bandwidth Approach 123
6.4.3 Optimization-based Approach 126
6.5 Comparison of Approaches 133
6.6 Summary 136

7 Self-adaptive Applications on Software Defined Infrastructure 139

7.1 Model 141
7.2 Implementation 149
7.3 Experiment 153
7.4 Summary 159

8 Conclusion 160

List of Tables

3.1 Specification of VMs in experiments on Amazon and SAVI clouds . . . 42
3.2 Experiment specification on SAVI cloud 42
4.1 Distribution of users in groups. 76
4.2 Experiment spec on SAVI and Amazon. VMs on SAVI are OpenStack small size. 76
4.3 Candidacy, Trigger, and SLA thresholds of response time for each application scenario. 77
5.1 Experiment specification on Amazon AWS 92
5.2 Parameters set for the scenarios in all experiments. The autonomic manger is to maintain the 95th percentile response time SLAs. 93
6.1 Symbols used in the optimization problem and their description. 113
6.2 Accuracy of machine learning models 116
6.3 Experiment spec on SAVI and AWS. VMs on SAVI are OpenStack small size. 118
6.4 Parameters 119

6.5 Three revenue models where $r_1 - r_4$ define price unit (e.g, \$) per request for Browse, Search, Auto-bundle and Pay respectively. 127
6.6 Comparison of auto-scaling and bandwidth adaptations in terms of time to effect. 133

7.1 Probabilities representing various adaptation requests 144
7.2 Exogenous parameters of the model measured on SAVI cloud 150
7.3 Experiment scenarios 151
7.4 Specification of VMs on SAVI cloud 153
7.5 Experiment specification on SAVI cloud 154

List of Figures

2.1 Four key components in typical feedback loops [23] 10

2.2 Autonomic Element 12

2.3 Overview of SDN Architecture 14

3.1 CAAMP Architecture: The traffic first arrives at the gateway in private cloud. If it looks normal, it will be redirected to the public cloud through the application proxy. Otherwise, it will be redirected to the shark tank that is dynamically created on private cloud. 34

3.2 Overlay network establishment in STO 38

3.3 SDN Controller Application Architecture 39

3.4 DDoS mitigation. Attack periods are shown in shaded background. CAAMP redirects the suspicious traffic to the shark tank on the fly. . . 43

3.5 SDN Controller Performance 48

3.6 Architecture and modules of the DevOps security framework. 51

3.7 Performance results of prototype implementation on AWS. 59

3.8 DSS utility metrics during the experiment. 60

4.1 Application autonomic manager and the managed system; autonomic manager manages the bandwidth and VMs. 67

4.2 Classification and declassification of scenarios based on the management policy; flows are classified into scenarios and receive a specific bandwidth. 69
4.3 An overlay network that spans over private and public cloud connects application components. The bandwidth rate of any link is programmed on the fly. 72
4.4 CPU utilization, no of web workers and bandwidth adaptation. Shaded areas represent the bandwidth actions in the three events. 79
4.5 Response time and arrival rate of scenarios during the experiment; the response time axis has been divided into three bands: (a) the white band indicates normal scenario response time (b) gray shows the trigger area and (c) orange area highlights the SLA violation area. 80

5.1 High-level architecture of the adaptive control mechanism. 88
5.2 Experiment setup on Amazon AWS. 92
5.3 Using the hill-climbing heuristic, bandwidth of Scenario 2 is adapted. . 95
5.4 Response time of scenarios in hill-climbing heuristic. 96
5.5 Using the model, bandwidth of Scenario 2 is adapted. 101
5.6 Response time and arrival rates in model-based. 102

6.1 High level architecture of the proposed solution 107
6.2 Implementation on hybrid cloud 117
6.3 Workload 120
6.4 **Auto-scaling-only:** CPU Utilization and no. of workers 121
6.5 **Auto-scaling-only** 122
6.6 **Model-based bandwidth approach:** CPU Utilization and no. of workers 124
6.7 **Model-based bandwidth approach:** Bandwidth rates 124

6.8 **Model-based bandwidth approach** 125
6.9 **Direct-ad:** CPU Utilization and no. of workers 127
6.10 **Direct-ad:** Bandwidth rates 127
6.11 **Direct-ad** 128
6.12 **Indirect-ad:** CPU Utilization and no. of workers 129
6.13 **Indirect-ad:** Bandwidth rates 129
6.14 **Indirect-ad** 130
6.15 **No-ad:** CPU Utilization and no. of workers 131
6.16 **No-ad:** Bandwidth rates 131
6.17 **No-ad** 132
6.18 Comparison of auto-scaling, model-based bandwidth approach (BW), and direct-ad optimization 136
6.19 Comparison of auto-scaling, model-based bandwidth approach (BW) and indirect-ad optimization. 137
6.20 Comparison of auto-scaling, model-based bandwidth approach (BW), and no-ad optimization. 138

7.1 Conceptual model of *transposition* and *networking-only* adaptations in SDI143
7.2 Model of adaptation scenarios on an SDI 147
7.3 Experiment setup on SAVI cloud 152
7.4 Packet response time 155
7.5 Application adaptation request (AR) response time 156
7.6 Application adaptation request (AR) response time: analytical model vs. experiment, Scenario C, $\lambda_f = 1000$flows/sec, $P_{nf} = 0.01$ 157

7.7 Application adaptation request (AR) response time at the SDN controller: analytical model vs. experiment 158

Co-Authorship

This thesis is based my published work listed below. In these publications, I am the main author and the major contributor of the research work presented. In particular, I contributed in the following ways except where noted: materializing the initial idea, researching related work, conducting experiments and analyzing the resulting data (developing required tools).

1. **Nasim Beigi-Mohammadi**, Mark Shtern, and Marin Litoiu , Adaptive Load Management of Web Applications on Software Defined Infrastructure, IEEE Transaction on Network and Service Management, October 2018 (Under revision).

2. **Nasim Beigi-Mohammadi**, M. Litoiu, M.Taba, L. Tahvildari, M. Fokaefs, E. Merlo, Vio Onut, A DevOps Framework for Quality-Driven Self-Protection in Web Software Systems, International Conference on Software Engineering and Computer Science, November 2018.

3. **Nasim Beigi-Mohammadi**, Mark Shtern, and Marin Litoiu , A Model-based Application Autonomic Manager with Fine Granular Bandwidth Control, 13th IEEE International Conference of Network and Service Management (CNSM), Tokyo, Japan, November 2017.

4. **Nasim Beigi-Mohammadi**, Hamzeh Khazaei, Mark Shtern, Cornel Barna, and Marin Litoiu, An Adaptive Service Management of Cloud Applications Using Overlay Networks, 2017 IEEE International Symposium on Integrated Network Management (IM 2017), Lisbon, Portugal, May 2017 **(Acceptance rate: 16%)**.

5. **Nasim Beigi-Mohammadi**, Hamzeh Khazaei, Mark Shtern, Cornel Barna, and Marin Litoiu, Implementation of Self-Managing Applications on Cloud Using Overlay Networks, 2017 IEEE International Symposium on Integrated Network Management (IM 2017), Lisbon, Portugal, May 2017

6. **Nasim Beigi-Mohammadi**, Cornel Barna, Mark Shtern, Hamzeh Khazaei, and Marin Litoiu , CAAMP: Completely Automated DDoS Attack Mitigation Platform in Hybrid Clouds, 12th IEEE International Conference of Network and Service Management (CNSM), Montreal, Canada, October 2016 **(Acceptance rate: 14.65%)**.

7. **Nasim Beigi-Mohammadi**, Hamzeh Khazaei, Mark Shtern, Cornel Barna, and Marin Litoiu, On Efficiency and Scalability of Software Defined Infrastructure for Adaptive Applications, IEEE International conference on Autonomic Computing (ICAC), Wurzburg, Germany, July 2016 **(Acceptance rate: 22.43%)**.

8. **Nasim Beigi-Mohammadi**, Marin Litoiu, Engineering Self-Adaptive Applications on Cloud with Software Defined Networks, 2017 IEEE International Conference on Cloud Engineering (IC2E) Doctoral Symposium, April 2017.

I also collaborated in following publications where in [1] I developed a simulation model to help with the understanding of the system for designing the analytical model. In [2], I collaborated in surveying NoSQL solutions:

1. Hamzeh Khazaei, Cornel Barna, **Nasim Beigi-Mohammadi** and Marin Litoiu, Provisioning Performance of Cloud Microservices Platforms, 8th IEEE International Conference on Cloud Computing Technology and Science (CloudCom), December 2016.

2. Hamzeh Khazaei, Marios Fokaefs, Saeed Zareian, **Nasim Beigi-Mohammadi**, Brian Ramprasad, Mark Shtern, Purwa Gaikwad and Marin Litoiu, How Do I Choose the Right NoSQL Solution? A Comprehensive Theoritical and Experimental Study, Journal of Big Data and Information Analytics (BDIA), December 2015.

Chapter 1

Introduction

Cloud computing enables application and service providers to quickly adapt to new technologies, increasing demands from customers, and a competitive business market. However, it still remains a challenge to meet non-functional requirements such as performance, cost, and security of applications in a volatile environment such as cloud. From performance point of view, it is not easy to maintain the Quality of Service (QoS) specified in the Service Level Agreement (SLA) while avoiding over provisioning in a volatile environment such as cloud [1]. Robust mechanisms should be in place to guarantee SLA compliance for different types of services. Application resources should be managed according to workload fluctuations and system uncertainties to maintain performance objectives.

With the cloud's "pay as you go" model, it is vital to consider economic aspects while managing the resources. Amazon web services (AWS) company, one of the major cloud providers, also emphasizes the importance of building software using "cost-aware architectures" that manage the cost of using cloud resources [2, 3].

Further, Denial of Service (DoS) attacks are one of the top nine threats to cloud

computing environment according to Cloud Security Alliance [4]; out of all attacks in cloud environment, 14% are DoS attacks. As cloud is becoming more widespread, the rate of DoS and Distributed DoS (DDoS) attacks is growing in parallel [5, 6].

To meet such application requirements, autonomic management systems (AMS) can be designed to continuously monitor applications' specific metrics, make decisions, and implement corrective actions when operating conditions change [7]. The most common approach to maintain performance at high load or partial failure is to leverage the cloud elasticity that provides on demand provisioning of computing/storage resources [8]. With mature virtualization technologies in computing, applications can easily adjust their computing/storage demands in cloud; resources are dynamically added to the application when there is a surge in the workload and they are removed when the workload drops. The AMS can be as simple as threshold based auto-scalers such as AWS auto-scaling [9] where resources are added/removed when certain conditions are met or as complicated as model-based auto-scalers such as [10, 11, 12, 13]. In threshold-based approach, for instance, when the average utilization of all of the scalable resources reaches a specific threshold, new resources will be added to the application cluster. The model-based auto-scaling may work in either reactive or proactive mode; similar to the threshold-based approach, in reactive model-based, when a certain condition is triggered, the model will determine the amount of needed resources and the location to add. In predictive mode, the model will predict these information ahead of time a priori the set condition.

Due to its automatic nature, auto-scaling can be exploited by those who want to gain money using other people's resources; cloud-based applications are now target of cryptocurrency mining attacks that steal compute resources. As a mining malware con-

sumes all available CPU power, the auto-scaler will automatically spawn new instances, allowing the miners to gain huge scalability at the expense of their victims. Recently a company's AWS bill went up from less than $10K to over $100K per month due to illegitimate mining traffic [14].

Also, auto-scaling can become a target of malicious intents where applications are scaled out due to sudden rise in suspicious and/or illegitimate traffic such as DDoS attacks that waste both CPU cycles as well as bandwidth of applications on the cloud - causing sever economic damage to the application owners [15]. The application is required to detect and mitigate attacks in seconds to prevent auto-scaling triggers that fire fleets of virtual servers coming online automatically, staying up for a period of time and then shutting down again when the load disappears. Application owners are left with unnecessary bills caused by mass over-provisioning.

Even when there is no financial or malicious purposes involved, sometimes the application is scaled out as a result of traffic that generate no or negligible revenue. In such cases, auto-scaling not only increases the operational cost but also leads to little or no revenue which reduces the overall profit. Therefore, it becomes important to take into account the trade-off between the cost versus the added revenue as a result of scaling as it can increase the cost highly disproportional to the corresponding revenue.

Furthermore, from efficiency point of view, scaling out the application, as the first adaptation strategy, imposes *reconfiguration cost*; when a new virtual machine (VM), for instance, is added to the application cluster, it will take a few minutes for the VM to be instantiated, which we refer to as *provisioning time*. In addition, the newly instantiated VM needs to be configured– and does not immediately participate in application service delivery, which we refer to as *contribution time*. Therefore, the new node will take

some time (provision time + contribution time) until its presence effects. Even in case of containerized applications, it will also take some time before the new container becomes fully operational [16]. In addition, in big data applications, scaling is not always desireable because of introducing challenges in data consistency, data replication, and job rescheduling.

With advent of software defined networking (SDN) and network function virtualization (NFV), it is now plausible to program the network on the fly. SDN separates the control plane from the data plane in networking devices and provides an API-driven management of the network. In NFV, network functions such as routing, switching, load-balancing, firewalls, and the like are virtualized and can be placed anywhere in the network dynamically. Given such network "softwareization", applications can take advantage of a fully programmable infrastructure, known as software defined infrastructure (SDI), to better meet their dynamic requirements. Provided that SDI opens new avenues for run-time adaptations that can help to maintain or enhance application non-functional requirements.

In this thesis, we investigate techniques and mechanisms to design application autonomic management systems that employ run-time compute and network programability within SDI to maintain performance, cost, and security objectives of applications in cloud. To the best of our knowledge, there is no prior work that takes advantage of both compute and network programmability at run-time to engineer self-adaptive applications in cloud. To achieve the research objectives, we strive to address the following questions:

1. How can we build autonomic management systems using *compute and network* adaptations to optimize the resource utilization while maintaining the performance

goals against application layer DDoS attacks?

2. How can we use *dynamic bandwidth allocation* that is planned at *the application layer* to meet performance criteria while minimizing the cost?

3. Is it possible to use a *model* as knowledge base to improve the quality of adaptation?

4. If yes, can we build a model that is *scalable* with respect to the application size?

5. How can we engineer a *comprehensive* autonomic management system that comprehends all of the above goals and *optimizes the overall profit*?

6. How to investigate the *efficiency and scalability of an SDI* to host such autonomic management systems that request for various adaptations at run-time?

We address each research question by a research contribution. To answer the first research question, we investigate techniques to design self-protecting applications in cloud to mitigate application layer DDoS attacks. To this end, we propose **C**ompletely **A**utomated DDoS **A**ttack **M**itigation **P**latform (CAAMP), a novel platform to mitigate DDoS attacks on public cloud applications using capabilities of SDI and network virtualization techniques. When suspicious traffic is identified, CAAMP deploys a copy of the application's topology on-the-fly (a shark tank) on an isolated environment in a private cloud. It then creates a virtual network that will host the shark tank. SDN controller programs the virtual switches dynamically to redirect the suspicious traffic to the shark tank until the final decision is made. If traffic is proved to be non-malicious, SDN controller installs flow rules on the switches to redirect the traffic back to the original application. CAAMP autonomously protects applications against potential DDoS

threats and lowers the false positives associated with common detection mechanisms by leveraging resources from a private cloud.

We then extend the idea of CAAMP using a Development and Operations (DevOps) approach to offer a mitigation solution that includes taking actions at the application layer as well as network layer to meet performance and cost requirement of the application.

To answer the second question, we propose a design, algorithm, and an implementation for a heuristic-based AMS that leverages overlay networks and SDN to dynamically control the bandwidth of application flows to meet the response time SLA of application scenarios. We show that our adaptive mechanism meets the performance goals and delays adding more resources for as long as possible to reduce the cost.

To answer the third question, we propose an on-line machine learning model that is able to predict right adaptation actions and overcome the limitation of the previously proposed heuristic-based approach.

To answer the froth and fifth questions, we propose a comprehensive AMS that takes into account the application requirements and performs adaptations that maximize the profit at run-time. We employ scalable machine learning models that are able to estimate bandwidth rates as well as performance metrics for an application with large number of scenarios (answering the forth research question). We develop an optimization model that considers various inputs such as multi-classes of workload and their impact on the revenue, cloud price model, and performance objectives to derive the best course of adaptations that maximizes the profit (answering the fifth research question).

To answer the sixth question, we study the efficiency and scalability of an SDI that host applications which perform run-time compute and network adaptations through

a queuing network model. Our analytical model yields the response time of realizing adaptations on the SDI and reveals the scalability limitations. Cloud service providers can leverage the proposed model to perform capacity planning and bottleneck analysis when they accommodate adaptive applications.

The proposed solutions in this thesis are all implemented in real clouds using state of the art technologies. The produced artifacts are the result of close collaboration with various industry partners such as Juniper Networks [17], IBM Center for Advanced Studies [18], CENGN [19], and TELUS [20] through extensive design camps, annual general meetings, workshops, and competitions.

The organization of this thesis is as follows: in Chapter 2, we provide background information including concepts, terms, and methodologies that are used throughout the thesis. Further, we overview related work and distinguish our work. We present self-protecting applications in Chapter 3. Chapter 4 presents the search-based AMS that adapts bandwidth rates of application scenarios to postpone adding resources to reduce the cost while meeting performance objectives. We present a machine-learning model to improve the quality of adaptation and speed up the run-time reaction in Chapter 5. In Chapter 6, we propose a comprehensive AMS that optimizes the profit, given the application requirements. Our proposed analytical performance model of SDI in presence of application adaptions is presented in Chapter 7. Finally, Chapter 8 concludes the thesis and states future work.

Chapter 2

Background and Related Work

In this chapter, we overview background materials including concepts and theories in Section 2.1. Following that in Section 2.2, we survey related literature and identify key differences between our work and related work.

2.1 Background

In this section, we overview background information. We fist overview adaptive systems. We then explain SDN and NFV. Later, main concepts in queuing networks and decision tree learning models are discussed.

2.1.1 Adaptive Systems

The dynamic nature of modern software systems has imposed new challenges to software engineers to enable such systems face various types of changes in terms of user requirements and environmental changes.

Self-adaptive techniques have been proposed to make software adaptable to changes.

Adaptation actions and policies are triggered proactively or reactively to allow a software system to offer acceptable levels of QoS, while preserving semantic correctness with respect to functional requirements. For example, customers may require continuous assurance of agreed performance indices such as response time that can be employed to trigger adaptations, guaranteeing that the requirements are met even in unforeseen environmental fluctuations [21].

Autonomic (self-adaptive) systems attempt to solve the problem of managing complexity through engineering certain properties, referred to as self-* properties (e.g. self-adapting, self-optimizing, self-tuning, self-healing, self-configuring, self-organizing and self-protecting) into the elements of a system. In a nutshell, an autonomic system possesses following characteristics [22]:

- Self-configuring: refers to the ability of a system to configure and reconfigure itself on the fly (e.g., installing, updating, integrating software entities).
- Self-healing: refers to the ability of a system to recover itself from disruptions and failures.
- Self-optimization: refers to the ability of a system to optimize its resource utilization without human intervention.
- Self protecting: refers to the ability of a system to protect itself from security breaches and malicious attacks.

However, to achieve the above-mentioned characteristics of an autonomic system, the system should be monitored and analyzed. If adaptation is required, it should be planned and executed. These processes are carried out by autonomic managers, which can be external to the system to be managed.

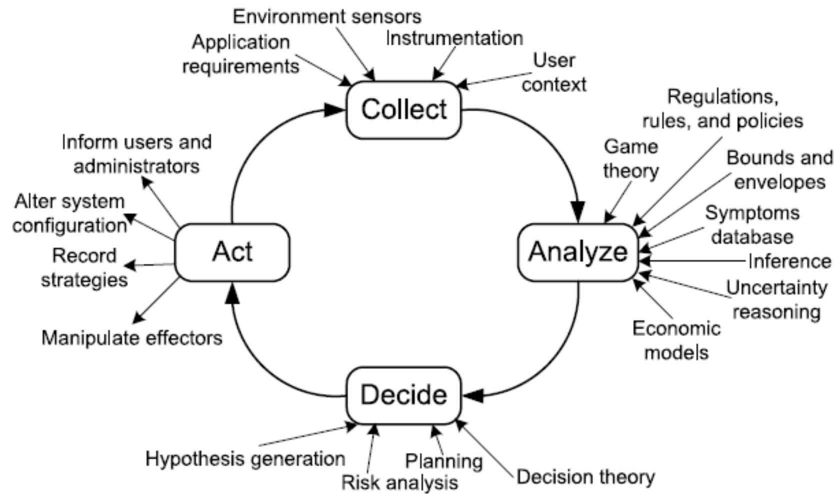


Figure 2.1: *Four key components in typical feedback loops [23]*

Feedback loops are essential parts of self-adaptation in adaptive systems. Self-adaptation is mainly performed through moving design decisions to run-time to manage dynamic behavior of the environment [7]. A feedback loop typically involves four key components including collecting, analyzing, planning, and executing as depicted in Fig. 2.1. Sensors collect data from the running system and its state. The collected data is then cleaned, filtered and stored to derive an accurate model of past and current state of the system. The resulted data is then analyzed to infer trends and diagnose problems. Subsequently, planner attempts to predict the future to decide on how to act on the system and its context through actuators or effectors. While the generic feedback loop (often referred to as autonomic control loop) presented in Fig. 2.1 provides a generic model of a feedback loop system, it does not provide the flow of data and control around the loop. In addition, multiple separate control loops can be applied in real world systems.

When engineering a self-adaptive system, there are a number of concerns that need

to be taken into account. First, in data collection phase, it is required to determine the required sample rate. Question arise about the validity of the sensor data. In addition, there has to be sufficient data about the system to provide more accurate system description. Second, many approaches exist on analyzing the collected data including models, theories and rules. Analyzing the collected data should infer the current state of the system. Also, it should be determined how much past state may be needed in the future. Other research questions include: What data needs to be archived for validation and verification? How faithful the model will be to the real world and whether the collected data can attain an adequate model? How stable will the model be over time?

After analyzing the state, decisions have to be made to achieve the desired state. Approaches such as risk analysis aid in choosing among various alternatives. Finally, to execute the decision, the system must act via available actuators or effectors. Questions arise regarding execution of the decisions including when should and can the adaptation be safely performed? How do adjustments of different feedback loops interfere with each other? Do centralized or decentralized feedback aid in attaining the global goal?

The above considerations regarding the feedback loops should be explicitly taken into account during the development of a self-adaptive system.

IBM introduced feedback loops explicitly in its autonomic computing initiative with its emphasis on engineering self-managing systems. More specifically, Kephart et al. [24] introduced autonomic element which was later used by IBM as an architectural blueprint for autonomic computing. The autonomic element is used as a reference model for autonomic control loops to manage autonomic systems. The autonomic element comprises two main parts including autonomic manager and managed element as shown in Fig. 2.2.

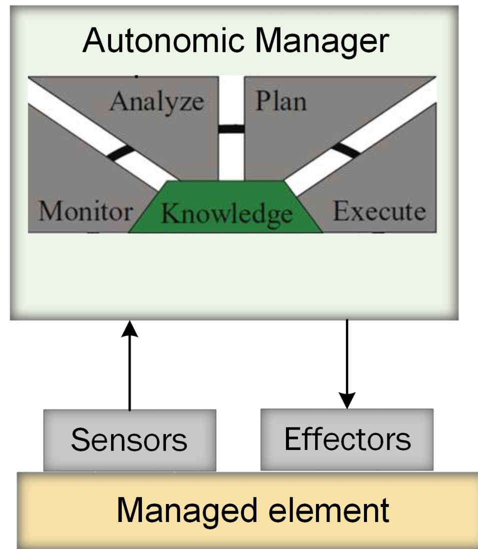


Figure 2.2: *Autonomic Element*

Autonomic manager implements the feedback loop and controls the managed element. The manager consists of two management interfaces: sensor and effector, and monitor-analyze-plan-execute (MAPE-K) engine composing of a monitor, an analyzer, a planner, and an executor that share a common knowledge base.

The monitor senses the managed process and its environment, filters the gathered data, and stores them in the knowledge base for future reference. The analyzer compares the gathered data with existing patterns in knowledge base to detect symptoms, and stores the symptoms in the knowledge base for future reference. The planner infers these symptoms and creates a plan to execute a change in the managed process through effectors. Hence, planning involves producing adaptation plans based on the monitored data from the sensors. The autonomic manager collects measurements form the managed element as well as the information about the past and the current states from the knowledge base to adjust the managed element if required.

2.1.2 Software Defined Networking (SDN) and Network Function Virtualization (NFV)

With the advent of technologies such as computing and storage virtualization as well as advanced cloud orchestration tools, SDN was a missing piece for building so-called ”software defined infrastructure” (SDI) [25]. These convergence enabled the emergence of fully programmable IT infrastructures. Four fundamental principles define SDN each of which is mandatory for a technology to be SDN:

- The control and data planes are separated. Control functionality is removed from network devices and they become simple (packet) forwarding elements.
- Forwarding decisions are made per flow basis. A flow is broadly defined by a set of packet field values that are used to identify flows in flow tables and a set of rules (instructions). In SDN, all packets belonging to the same flow will be treated equally [26].
- In the context of SDN, controller is called network operating system (NOS) as it acts as the brain to the network and it is run on commodity server technology. It provides the essential resources and abstractions to facilitate the programming of forwarding devices based on a logically centralized, abstract network view. Its purpose is therefore similar to that of a traditional operating system.
- The network is programmable through applications that are run on top of the NOS and interacts with the underlying data plane devices.

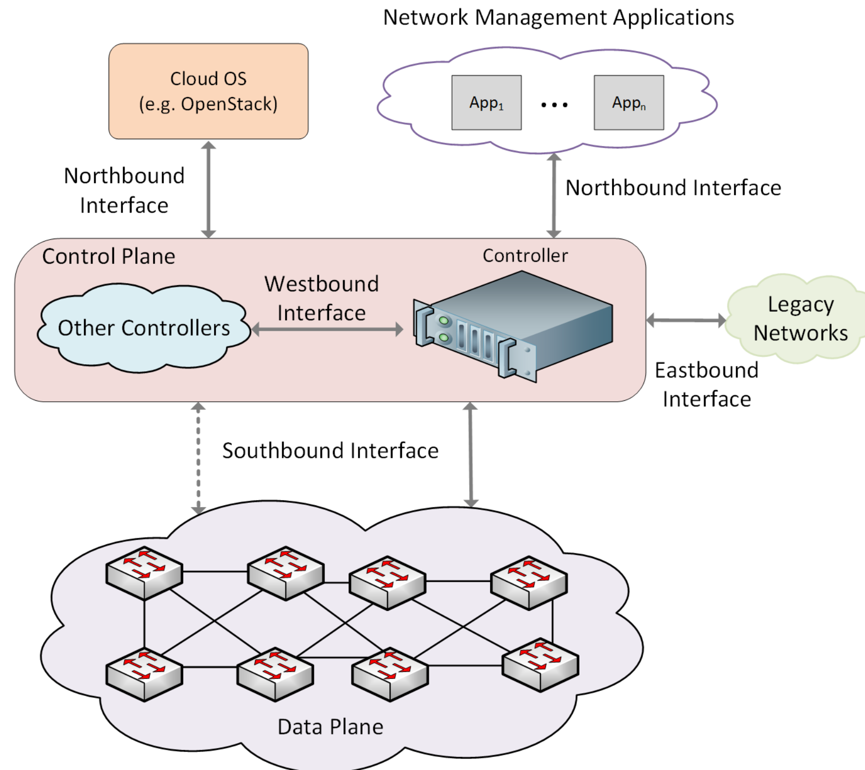


Figure 2.3: *Overview of SDN Architecture*

SDN Components

Figure 2.3 shows architecture of SDN where the controller and the forwarding devices interact via a well-defined API and the controller exercises direct control over the state in the data plane elements via such an API. Major component of SDN are:

- **Forwarding devices:** includes switches and routers that contains the flow tables installed by the controller and forward the packets based on the flow tables. In SDN terminology, such forwarding devices are commonly referred to as switches. Hereafter we refer to forwarding devices as switches. In SDN, switches come in two types: pure and hybrid. Pure SDN switches have no legacy features or on-board

control, and completely rely on a controller for forwarding decisions. On the other hand, hybrid switches are SDN-enabled in addition to traditional operation and protocols.

- **Data plane:** refers to the network of switches that are connected using wired or wireless technologies.

- **Controller:** is the control plane of SDN whose applications define the rules that are installed on the switches.

- **Southbound API:** The instruction set of the switches is defined by the southbound API as one part of southbound interface. Another part of southbound interface is the communication protocol through which messages are exchanged between controller and switches. The mostly accepted standard for southbound API is OpenFlow [27] that was originally implemented at Stanford University. Currently OpenFlow is considered the main SDN southbound API standard.

- **Northbound API:** provides the interface for the controller and the application running over the controller. Unlike controller-switch communication, there is no currently accepted standard for northbound interactions and they are more likely to be implemented on an ad hoc basis for particular applications [28]

- **Westbound/Eastbound interfaces:** for scalability and availability purposes, there may be multiple controllers in a distributed fashion that communicate and share network policy information through northbound interface. Unlike southbound interface, there is no currently accepted standard for westbound/eastbound interactions, however, SDNi [29] specifies common requirements to coordinate flow setup and exchange connectivity information across multiple domains.

- **Management Plane:** is the set of applications that implement network control and operation logic. This includes new or legacy network functions such as routing, firewalls, load balancers, monitoring, etc. Essentially, such applications will be translated to instructions sent to the switches via southbound interface [30].

Network functions can be virtualized on virtual machines or containers that can be placed dynamically anywhere in the network to meet high-level application objectives. This approach is referred to as network function virtualization (NFV). Network functions includes switching, routing, load balancing, firewall and the like. NFV removes the need for specialized middle-boxes as they can be virtualized on commodity hardware.

2.1.3 Queuing Networks

Queuing network modelling is a particular approach to computer system modelling in which the computer system is represented as a network of queues that is evaluated analytically. A network of queues is a collection of service centers, which represent system resources, and customers, which represent users or transactions. There are three types of queuing networks: open, closed an mixed networks. In open queuing networks there are external arrivals and departures while in closed networks there is a constant number of customers (i.e., finite population). If the system is open for some customers and closed for others, it is a mixed network.

Each queuing network is defined by a set of service centers, customers, and topology. Each service center is defined by the number of servers, their service rate, and the queuing discipline. Custmores are described by the arrival process to each service center for open queuing networks, their number for closed queuing networks and the service demand to each service center. Service demand is expressed in units of service /units

of time. Service time is equal to service demand/service rate which is a non-negative random variable with mean denoted by $1/\mu$.

The network topology models the customer behavior among the interconnected service centers. Routing probability defines how customers move from one station to another or leave the network. There may be different types of customers each of which are defined by their required demand and routing probabilities.

Analysis of Queuing Systems

Queuing networks consist of queuing systems where there is a single service center in each queuing system. Kendall's notation $(A/B/X/Y/Z)$ is used to describe a queuing system where

- A is the arrival rate distribution (λ);
- B is the service rate distribution (μ);
- X is the number of servers (n);
- Y is the system capacity (in the queue and in service);
- Z is the queuing discipline.

When the queuing system is defined as A/B/X, we assume $Y = \infty$ and $Z =$ First Come First Serve (FCFS). There are two types of analysis that can be performed on queuing systems [31]. Transient analysis is where the performance indicies are calculated for a time interval, given initial conditions. Stationary analysis is performed for steady state conditions for stable systems. Performance indices include:

- n: number of customers in the system

- w: number of customers in the queue
- r: response time
- t: waiting time
- U: utilization
- X: throughput

The result of performance analysis can be random variables presenting performance indices in a form of probability distribution or it can be in average format such as mean response time.

M/M/1 Qeueing System: is a basic qeuing system where the arrival rate (λ) follows Poisson process, the service time is exponentially distributed ($1/\mu$) and there is a single server. The condition for stability is held when $\lambda < \mu$, then other performance indices using Little's law can be calculated as:

$$\rho = \lambda/\mu \tag{2.1}$$

$$R = 1/(\mu - \lambda) \tag{Little's law}$$

$$X = \lambda \tag{2.2}$$

$$U = \rho \tag{2.3}$$

$$W = \rho^2/(1-\rho) \tag{2.4}$$

$$T = (1/\mu)(\rho/1-\rho) \tag{Little's law}$$

Where ρ is traffic intensity, N is the mean number of customers, R is the mean response time, W is the mean queue length, and T is the mean waiting time.

We use a network of $M/M/1$ queues to model an SDI in presence of application adaptation requests in Chapter 7.

2.1.4 Decision Tree Learning Models

Decision Trees (DTs) are a non-parametric supervised learning method used for classification and regression. The goal is to create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features. A tree can be learned by splitting the source set into subsets based on an attribute value test. This process is repeated on each derived subset in a recursive manner called recursive partitioning. The recursion is completed when the subset at a node has all the same value of the target variable, or when splitting no longer adds value to the predictions. This process of top-down induction of decision trees is an example of a greedy algorithm, and it is by far the most common strategy for learning decision trees from data.

There are two types of decision trees: classification trees and regression trees. Classification trees are used when the predicted value is a class while regression result is a real number. Trees used for regression and trees used for classification have some similarities - but also some differences, such as the procedure used to determine where to split.

There are many specific decision tree algorithms. Notable ones include:

- ID3 (Iterative Dichotomiser 3)
- C4.5 (successor of ID3)
- CART (Classification And Regression Tree)
- CHAID (CHi-squared Automatic Interaction Detector). Performs
- multi-level splits when computing classification trees.[11]
- MARS: extends decision trees to handle numerical data better.

In this thesis, we use CART regression algorithm in Chapters 5 and 6 to estimate a number of performance indices. Following we explain the mathematical formulation behind it.

2.1.4.1 CART Algorithm

Given training vectors $x_i \in R^n$, $i = 1, \ldots, l$ and a label vector $y \in R^l$, a decision tree recursively partitions the space such that the samples with the same labels are grouped together. Let the data at node m be represented by Q. For each candidate split $\theta = (j, t_m)$ consisting of a feature j and thershhold t_m, partition data into $Q_{left}(\theta)$ and $Q_{right}(\theta)$ subsets.

$$Q_{left}(\theta) = (x, y) | x_j <= t_m \tag{2.5}$$

$$Q_{right}(\theta) = Q \setminus Q_{left}(\theta) \tag{2.6}$$

The impurity (G) is calculated as

$$G(Q, \theta) = n_{left}/N_m H(Q_{left}(\theta)) + n_{right}/N_m H(Q_{right}(\theta)) \tag{2.7}$$

Select the parameters than minimizes the impurity

$$\theta^* = arg\, min_\theta\, G(Q, \theta) \tag{2.8}$$

Impurity function, $H()$, can be calculated as Mean Squared Error (MSE), which minimizes the L2 error using mean values at terminal nodes, and Mean Absolute Error (MAE), which minimizes the L1 error using median values at terminal nodes as below:

$$y_m = 1/N_m \sum_{i \in N_m} y_i \tag{2.9}$$

using MSE

$$H(x_m) = 1/N_m \sum_{i \in N_m} (y_i - (\bar{y}_m))^2 \tag{2.10}$$

or MAE

$$H(x_m) = 1/N_m \sum_{i \in N_m} |y_i - (y_m)| \tag{2.11}$$

where N is the number of observations at node m and x_m is the training set at node m.

Recurse for subsets $Q_{left}(\theta^*)$ and $Q_{right}(\theta^*)$ until the maximum allowable depth is reached, or $N_m < min_{samples}$ or $N_m = 1$.

Mechanisms are used to prevent creating over-complex trees that do not generalize the data well which is called over-fitting. Hyper parameters such as pruning, setting the

minimum number of samples required at a leaf node or setting the maximum depth of the tree are necessary to avoid this problem [32].

2.2 Related Work

In this section, we overview related work that includes application-awareness in SDN and SDN performance, DDoS mitigation, and auto-scaling and load management in cloud. In each part, we elaborate how our work is different from or built upon related work.

2.2.1 SDN: Application Awareness and Performance

With the emergence of SDN, cloud providers can expose APIs to cloud users so that users can manage their networking resources the same way they manage their computing and storage resources [33, 34]. However, most public cloud users do not have access to the cloud SDN APIs, if there is any [35]. Wickboldt et al. [34] propose a design of a cloud platform that puts network on the same level with computation (CPU) and storage (disk) resources; this way the client applications can dynamically provision and de-provision network as needed.

The Smart Applications on Virtual Infrastructure (SAVI) project [36] was established to investigate future application platforms designed for practicing software defined projects using OpenStack [37]. We use SAVI testbed to validate our solutions.

Run-time adaptations to accelerate the performance of big data applications have been explored to reduce the job completion time [38, 39]. The solutions presented in [38, 39] particularly study the run-time network configuration for big data applications

to jointly optimize application performance and network utilization. They focus on Hadoop as an example to explore the design of an integrated SDN control plane and describe Hadoop job scheduling strategies to reduce time-to-insight.

Load balancing is a commonly used technique to distribute a workload between nodes through which system availability is ensured. Handigol et al. [40] propose load balancer Plug n Serve for unstructured networks that tries to reduce the response time by taking into account the computing capacity of the servers as well as the congestion of the network. Given the load of the servers and network congestion, the controller application decides where to direct the traffic. This way servers are dynamically provisioned to the network. Wang et al. [41] design a load balancer similar to Plug n Serve but in a proactive approach based on wild cards. Wild cards can be used to aggregate the client requests based on the ranges of IP prefixes. In [41], they balance the load from monitoring the network traffic in a proactive manner without the intervention of controller as opposed to Plug n Serve which takes into account the load of the servers.

Most of the related work use cloud provider network to make dynamic networking configurations. However, our cloud agnostic solution takes advantage of overlay networks on top of cloud provider network that brings about higher flexibility and control to applications to manage their networking. Our proposed management mechanism is a versatile solution where an autonomic manager continuously monitors the response times of distributed cloud applications and make corrective actions as needed by communicating to SDN controller. Using model-based adaptive bandwidth management, our proposed mechanisms manage to maintain SLA while postponing adding extra resources which is the common approach for applications on the cloud.

To verify the performance of SDN, extensive simulations and experiments have

been performed. Such studies span from using Mininet emulator [42] to experimental testbeds. Turull et al. [43] evaluate the performance of an OpenFlow network through Mininet simulations. They compare three network virtualization technologies including Flowvisor-Open Virtual Switch (OVS), Slicable Switch by Terma and LibNetVirt. They investigate how delay of switch-controller interactions affect the ICMP, TCP and UDP traffic. Studies such as [43] mostly depend on the application they consider, many of which use simple learning switch to measure the throughput of SDN. Although simulations and experiments are popular performance evaluation techniques, they are costly and take time to obtain results. Analytical modeling, on the other hand, can quickly provide performance indicators and potentially bottlenecks of SDN in a cheaper and faster way. There are only a few works on the analytical modeling of SDN and it requires higher attention from research community as also pointed out in [30]. Following we discuss the related work on analytical modeling of SDN.

Jarschel et al.[44] derive a basic model of SDN using a feedback queuing system. The switch and controller are modeled by $M/M/1$ and $M/M/1/S$ queue systems, respectively. They measured system parameters including the service time of the controller through experimental setup and introduce such parameters to their analytical model.They focus on the response time of SDN in processing flow arrivals. However, they do not fully reflect the feedback from the controller to switch in their modeling. Yao et al. [45] model a controller using $M^k/M/1$ where flow setup requests from the switches arrive to the controller queue. They analyze the controller performance in terms of average flow service time. They further extend their model to multiple controllers. Their model, however, is not complete because they did not consider the whole switch-controller interactions.

A network calculus-based approach is used to quantify the packet processing capability of the switch in [46]. However the feedback between the controller and switches is not considered. Mahmood et al. [47] model SDN using a Jackson queuing system where they tailor the parameters to fit a SDN system into a Jackson model. They extend the model to include more than one switch per controller.

The aforementioned related work does not consider performance of SDN in the context of application-awareness in SDIs, in addition to the mentioned limitations. Hence, our work differs from related work in presenting an analytical model that takes into account not only the controller-switch interactions but also the ability of the application to configure network flows at run-time. We consider scenarios of application adaptations that require run-time computing and networking changes. Furthermore, most of the related work on SDI and SDN in particular, use simulators or emulators such as Mininet [42] for validation, however, we validate our analytical model through experiments on real clouds. Our cloud agnostic solution takes advantage of overlay networks on top of cloud provider network that brings about higher flexibility and control to applications to manage their networking.

2.2.2 DDoS Mitigation

Arbor Networks [48] surveyed how DDoS attacks evolved over the past decade, and the current threats that data centers have to handle. The survey found that the largest attack in 2014 has peaked at 400 Gbps, and more than 75% of data centers were targeted. It also reveals that the number of attacks and their intensity continue to increase. Barna et al. [49][50] investigate mitigation techniques at the application level. They use a performance model to analyze the impact of the traffic on some key performance metrics.

The traffic that is identified as undesirable is redirected (using the HTTP redirect header) to a secondary system where the users are challenged with a CAPTCHA. The requests that are part of a DoS attack will be discarded (since the CAPTCHA is not solved) and the misidentified requests are going to be slowed down (but not dropped). These results are further extended in [51], where the authors also use cost of resources when the decisions are made. Kalliola et al. [52] use machine-learning-based DDoS defense mechanism and use SDN techniques to maintain service quality by detecting and blocking the attacks upon detection. They evaluate their defense strategy using testbed experiments. Wang et al. [53] propose a DDoS attack detection and reaction solution. In their work, the DDoS mitigation strategy depends on public cloud provider to take actions against threats. In addition, Wang et al. [53] do not define where the suspicious traffic is hosted after being detected. Fung et al. [54] propose a dynamic traffic engineering solution to perform DDoS mitigation by assigning suspicious traffic to lower priority tunnels. They evaluated their solution using simulation.

Our work is different from above work on several dimensions: in our solution, we do not block suspicious traffic once it is detected; instead, we dynamically scale out and create a shark tank which offers the same service as the target application on private cloud; CAAMP is an autonomic and versatile solution that isolates the suspicious traffic to protect the application and creates the environment for further analysis of traffic. Moreover, by leveraging resources from private cloud, our solution attains a cost-effective mitigation strategy. We evaluate our solution on a real environment using Amazon as public cloud and SAVI as private cloud.

2.2.3 Auto-scaling

There is a whole body of research that uses auto-scaling to deal with workload fluctuations to continuously meet service level objectives.

Han et al. [55] apply multiple combinations of utilizations and SLA violations to reach a decision about scaling. Amazon auto-scaling [9] allows users to build scaling plans for various resources to maintain certain levels of utilization at any given time.

Evangelidis et al. [11] use discrete-time Markov chain (DTMC) as probabilistic modeling to model the dynamics of the auto-scaling policies. They try to improve the limitations of reactive, rule-based auto-scaling policies by computing the probabilities of CPU utilization and response time violation for each auto scaling policy passed as an input to the model. Wu et al. [56] propose a customer driven SLA-based resource provisioning algorithms to minimize cost by minimizing resource and penalty cost and improve customer satisfaction level by minimizing SLA violations.

Fan et al. [57] propose an adaptive resource scheduling strategy that can be used for calculation of run-time of each job to reduce the number of invoked virtual machines. They use a reflection mechanism to construct the resource scheduling model of cloud. Computation Tree Logic (TLC) is used to describe the properties of resource scheduling model. An adaptive resource scheduling dynamically re-optimizes and re-distributes resources at run-time.

Iqbal et al. [58] use a workload prediction that is used for future resource demand prediction and proactive auto-scaling to dynamically control the provisioning of resources. Huang et al. [59] present a resource prediction model for CPU and memory utilization based on double exponential smoothing, and compare it with simple mean and weighted moving average. Jiang et al. [12] use history data to predict request arrival rates using

linear regression and time series analysis. They use queuing theory to model the system and a multi-objective optimization to find the optimal number of virtual machines. Shariffdeen et al. [60] propose a proactive and cost-aware auto-scaling solution by combining a predictive model, cost model, and a smart killing feature. They also use time series and machine learning models to predict future workload.

Nikravesh et al. [61] use support vector machine and neural networks for the workload prediction using time series analysis for cloud resource provisioning. Benifa et al.[10] use a reinforcement learning approach (RLPAS) for resource allocation in cloud environment. RLPAS follows a parallel learning procedure for constructing optimal policies. They combine the strengths of parallel learning and function approximation to reduce the state space and to attain a faster convergence rate. Such predictive approaches should predict future workloads over a reasonable time to be effective. They should be also equipped with online training capabilities to learn new workload patterns [62]. Time-series method can also be combined with reactive methods such as [63, 64] to accomplish auto-scaling decisions.

Barna et al. [65], proposed *Hogna*, an autonomic manger that performs auto-scaling adaptations to maintain service level objectives. Hogna uses Opera [66] that is a qeueing network model of multi tier web applications to predict the number of needed VMs to add or remove. We use Hogna to build the application cluster and auto-scaling adaptations. Gandhi et al. [13] propose Dependable Compute Cloud (DC2), that automatically scales the infrastructure to meet the performance requirements of applications. DC2 makes use of Kalman filters to automatically learn the system parameters for each application to proactively meet application objectives.

Baresi et al. [67] present an autoscaling technique that allows containerized appli-

cations to scale their resources both at the VM level and at the container level. They use a Grey-box discrete-time feedback control as the planner. Iqbal et al. [68] utilize polynomial regression to model the number of servers in a tier as a function of the number of static and dynamic requests received by the RUBiS web application.

Related work does not consider the workload types and its impact on the profit. In other words, the trade-off between the revenue added and the cost associated with auto-scaling with respect to the workload have not been accounted for. Also auto-scaling is the only option considered to meet application requirements. We proposed to use dynamic bandwidth allocation as a new adaptation possibility.

2.3 Summary

In this chapter, we overviewed main concepts, theories and methods that are used throughout the thesis. We surveyed related work and differentiated our work from related work.

Chapter 3

Self-protecting Applications on Software Defined Infrastructure

Denial of service attacks impose a serious threat on applications in cloud [69]. Although there are benefits to dynamic resource provisioning with pay-as-you-go billing in cloud, one disadvantage is the possibility of attacks designed to increase the costs without corresponding increase in the benefit of the application. Distributed denial of service attacks (DDoS) is also one of the top nine threats to cloud computing environment according to Cloud Security Alliance [4]; out of all attacks in cloud environment, 14% are DoS attacks. As cloud is becoming more widespread, the rate of DDoS attacks is growing in parallel [5, 6]. DDoS attacks increase the workload on applications which would lead to adding more computational power to withstand the additional load. That is why DDoS attacks on scalable cloud resources are called Economic Denial of Sustainability (EDoS) attacks [70] as they cause economic damages because of unnecessary scaling. In addition, application owners are charged significantly for bandwidth usage caused by DDoS flooding, raising the bill for cloud usage drastically. Example of such at-

tacks include Low-and-slow denial of service (LSDoS) attacks which are low-bandwidth application-layer DoS attack. The attacker is able to slowly degrade the performance of the application by exploiting weaknesses they have identified [71].

Many companies use hybrid cloud deployment to allow them to scale computing requirements beyond their infrastructure capacity and still use their private cloud to reduce the cost. In addition, the private cloud can be used to host sensitive data. A hybrid cloud environment also gives rise to new opportunities to defend against DDoS attacks. In this chapter, we propose mitigation strategies that take advantage of a multi-cloud setting as in hybrid cloud to defend against DDoS attacks. We first present **C**ompletely **A**utomated DDoS **A**ttack **M**itigation **P**latform (CAAMP) [1], a fully software defined solution that is cloud agnostic due to its virtualized and modular design in Section 3.1. The mitigation in CAAMP is planned at the application level and implemented at the infrastructure level. Resources from private cloud are used to handle suspicious traffic so that more time can be bought to distinguish attacks from non-malicious transient behavior. To achieve its objectives, CAAMP makes use of SDN and network virtualization techniques as well as an autonomic manager which monitors the target application and initiates the mitigation process.

Second in Section 3.2, following CAAMP architecture, we propose a development and operation (DevOps) framework for security adaptation that is planned at the application level and implemented at application and infrastructure levels. The proposed solution enables the development and operations teams to collaborate and address security vulnerabilities. The proposed framework spans across the different phases of software (development, operations, maintenance) and considers all other factors (per-

[1] The name is inspired by Completely Automated Public Turing test to tell Computers and Humans Apart (CAPTCHA), whereas CAAMP helps to distinguish between malicious and non-malicious users by taking suspicious traffic to an isolated environment for further analysis.

formance, cost, functionality), when deciding for security adaptations. We demonstrate the approach to show how teams work together to tackle security concerns.

The contributions of this chapter are summarized as below:

- We introduce and implement CAAMP which is an effective mitigation platform that protects applications against severe damages of DDoS attacks by taking advantages of virtualization and SDN technologies. CAAMP allows to have a more accurate detection consensus by providing more time for traffic analysis while maintaining service availability in case suspicious traffic is not malicious. CAAMP is cloud agnostic and can be applied to any cloud environment. Using the resources from an existing private cloud, CAAMP bears a cheap solution in identifying and isolating real attacks.

- We evaluate CAAMP on real hybrid cloud environment where AWS is used as the public cloud and SAVI cloud [36] is leveraged as the private cloud. Results reveal that CAAMP is an effective and versatile solution that can protect applications on the cloud against attacks.

- We extend CAAMP with a game-theoric planner to offer a mitigation solution that includes taking actions at the application layer as well as network layer to meet performance, cost and functionality requirements of the application. We demonstrate the solution through experiments on real cloud.

3.1 CAAMP

Figure 3.1 shows the architecture of CAAMP. When suspicious traffic is detected, CAAMP provisions a copy of the original application on a private cloud, referred to

as *shark tank*, and dynamically redirects the suspicious traffic there. CAAMP protects stateless applications such as web applications where the transition to/from the shark tank does not interfere with user data. The shark tank provides the same functionalities as the original application, though with minimum amount of resources. The advantage of CAAMP to mitigate DDoS attacks is multi-folded; (1) DDoS attacks are hard to detect. Due to large false positives, non-malicious users may be filtered out as well [72]. Therefore, by using a shark tank, more time can be spent for further analysis over the suspicious traffic to make sure that the traffic is actually malicious and it is not some transient eccentric behavior. (2) By isolating the suspicious traffic on the shark tank, the original application is protected and serves only the traffic deemed legitimate. The users will experience normal behavior of the application and will not feel the negative effects of the attack. (3) The attacker that is sent to the shark tank has no indication that they have been discovered; they will experience high response time and the attack will look successful. As a consequence, they will not feel the need to adapt their strategy. (4) In case of adaptive applications, DDoS attacks may lead to excessive resources consumption (e.g., adding new virtual machines, increasing bandwidth, etc.) which will lead to higher costs. Therefore, by taking advantage of resources on a private cloud to host the shark tank, only the minimum amount of resources is allocated to handle suspicious traffic and reduce the associated costs.

Main elements of CAAMP consist of target application, DDoS sensors, shark tank, an autonomic manager, shark tank orchestrator, and an SDN controller which will be described below.

Target Application: CAAMP protects the target application using a hybrid cloud environment where the target [2] application is hosted on a public cloud. Tailoring

[2]We use target and original application interchangeably hereafter.

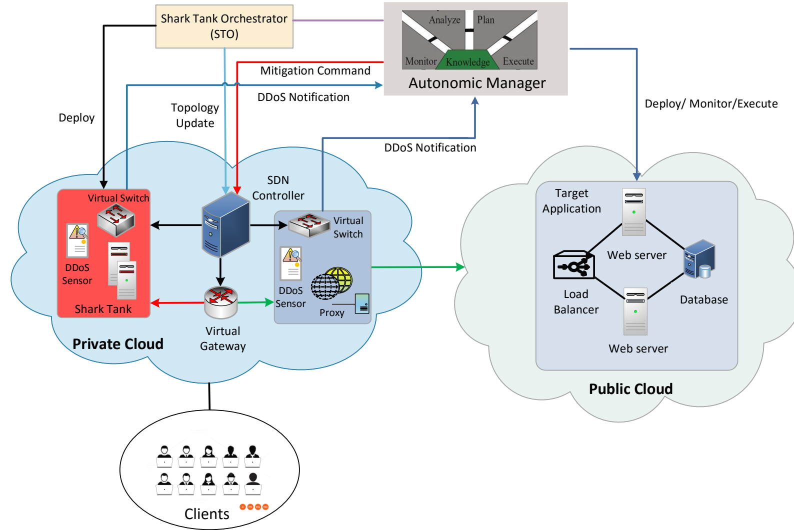


Figure 3.1: *CAAMP Architecture: The traffic first arrives at the gateway in private cloud. If it looks normal, it will be redirected to the public cloud through the application proxy. Otherwise, it will be redirected to the shark tank that is dynamically created on private cloud.*

CAAMP to work on a pure public or private cloud would be a trivial task.

DDoS Sensors: The objective of this component is to detect the existence of DDoS threats. DDoS sensors are used on all traffic: *original application traffic* and *shark tank traffic*; in former they initially detect the suspicious traffic which leads to redirection to the shark tank, and in latter they determine if the suspicious traffic redirected to shark tank is actually malicious or not. If traffic is not malicious, it will be redirected back to the original application. In case of malicious traffic, two strategies can be taken: (1) the malicious traffic is kept in the shark tank so that the attacker feels that the attack has been successfully completed and as a result does not change its strategy, (2) simply block the attack.

Shark Tank: The purpose of a shark tank is to act as a restricted area for the attackers so that they can be placed under close surveillance. The shark tank absorbs the damage from the DDoS attacks (i.e., all suspicious traffic will be redirected to one

instance of application in the shark tank) and allows the system to learn from these attacks for future reference. Moreover, the shark tank will be constantly monitored to investigate if the attacks are still taking place. The shark tank aids to reduce false positives, that is, if by mistake a user is redirected to the shark tank, they still get the requested services. Another reason for using a functional copy of the actual application on the shark tank is to prevent the shortcomings of the honeypot (the typical approach), that is, the malicious user might learn that they have been detected after they discover the honeypot has limited interactivity (simulation of the protected application). Although the same service is being offered, the amount of resources that will be used in the shark tank will be less than the original application.

Autonomic Manager: The autonomic manager uses a Monitor-Analyze-Plan-Execute (MAPE) loop [73] to monitor the protected application, plan and execute corrective actions so that desired operation conditions are met. For this purpose, CAAMP uses Hogna platform [65] to attain application inputs when creating the shark tank. In addition, Hogna monitors the application and adds or removes virtual machines (VMs) to meet the application criteria. For more detail on Hogna, reader is referred to [65].

SDN controller: It controls the routing of the traffic. Depending on the condition of the traffic, it programs the flow rules on the switches either to the target application or to the shark tank dynamically. The details and architecture of the SDN controller application is discussed in section 3.1.1.1.

Shark Tank Orchestrator (STO): It deploys the shark tank on private cloud for each application using *overlay networks*; this is one of the key points in making our solution cloud-agnostic. Particularly, STO employs Virtual Extensible LAN (VXLAN) technology which is a tunneling technology that can create arbitrary Layer 2 (L2) or

Layer 3 (L3) networks by encapsulating L2 frames in L3 UDP packets. We employ Open Virtual Switch (OVS) [74] and OpenFlow initiatives to set up the overlay networks and program the flows respectively. Details of STO operation is discussed in Section 3.1.1.

CAAMP Workflow

Initially, CAAMP brings up a virtual switch that acts as an on-premise gateway on application owners' private cloud. The clients' traffic first reaches the gateway and then will be redirected to the public cloud through the application proxy (See Figure 3.1). DDoS sensors are installed on proxy to sniff the incoming traffic and detect suspicious activity. If suspicious traffic is detected, the DDoS sensors report the incidents to autonomic manager. The autonomic manager fires up the mitigation process resulting in the creation of shark tank. It starts by instructing the STO to bring up the shark tank, collects the IDs and IPs of the new VMs and configures them for their respective roles (i.e., database, web workers). Then, the autonomic manager will send the commands to the SDN controller so that the required rules are installed on the switches to dynamically redirect the incoming suspicious traffic to the shark tank. The autonomic manager also installs DDoS sensors on the shark tank so that they continuously monitor the suspicious traffic. When DDoS sensors determine that the suspicious traffic is not malicious and it has been some transient behavior, the traffic is sent back to the original application. In case of attacks, if the mitigation policy is to keep the attack in the shark tank, no further commands will be sent to the SDN controller. Otherwise, the autonomic manager sends commands to the SDN controller to block the attacks.

3.1.1 Implementation

STO creates the shark tank environment dynamically based on a *topology descriptor* file that is provided by the autonomic manager. The topology descriptor file determines the shark tank specifications including the number of VMs, corresponding images, and the topology.

STO uses virtual switches to build any type of topology over L2 or L3 networks. The image used to instantiate the VMs can be pre-configured with the required services. For example, a VM that is going to host the web application may use a pre-configured image in which a web server service such as Apache Tomcat has been already installed. When topology information is provided, STO starts to provision the shark tank. We developed STO such that it sends topology information to the SDN controller while the shark tank is being provisioned. Hence, the SDN controller is quickly updated with the latest network topology. Thus, three main steps are involved in provisioning the shark tank:

1. VM instantiation: when topology descriptor file is provided to STO, STO starts instantiating the required VMs based on the specified images. The specified images contain the required applications and services.

2. Overlay Network Establishment: after the VMs are up and running, the VXLAN links are used to build the overlay networks and construct the topology. OVS software is pre-configured on the images and a script issues commands to each VM to establish VXLAN tunnels. This process is performed at two levels: host level and switch level. At the host level, the script issues commands on each host to connect them to their corresponding virtual switches. At the switch level, VXLAN tunnels are constructed to connect the switches to create the desired topology. Fig.

3.2 illustrates the overlay network establishment in private cloud corresponding to Fig. 3.1. STO creates different overlay networks to ensure traffic separation between the networks. The reason for creating overlay networks only on private cloud is because after detection, only the safe traffic will reach the public cloud (i.e., mitigation happens before traffic reaches public cloud.)

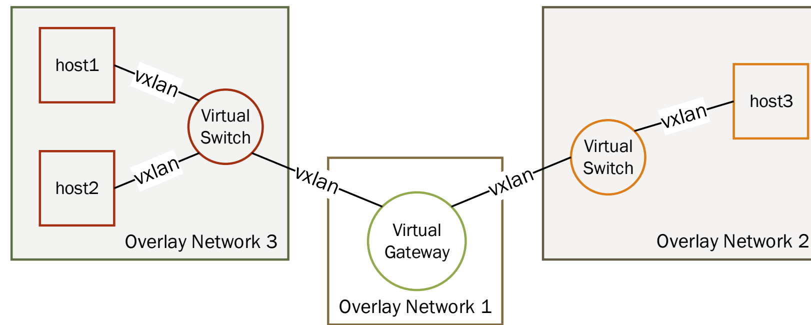


Figure 3.2: *Overlay network establishment in STO*

3. Topology Update: whenever there is a change in topology (adding/removing VMs, switches, links etc.), the changes are dynamically sent to the SDN controller. The details of this process is explained in Section 3.1.1.1.

3.1.1.1 SDN Controller Design

The SDN controller acts as the brain of the network and it should always hold a global view of the network so that it is able to configure the switches properly [75]. Our SDN controller runs Ryu SDN framework [76] and programs the network flows using OpenFlow. Our SDN controller does not need a topology discovery module as opposed to other SDN controllers such as Floodlight [77] that involves overhead of topology

discovery. Instead, when STO creates the topology, it sends the topology information to the SDN controller so that the SDN controller knows the topology a priori. Following, we explain how the topology information and mitigation actions are sent to the SDN controller in more details.

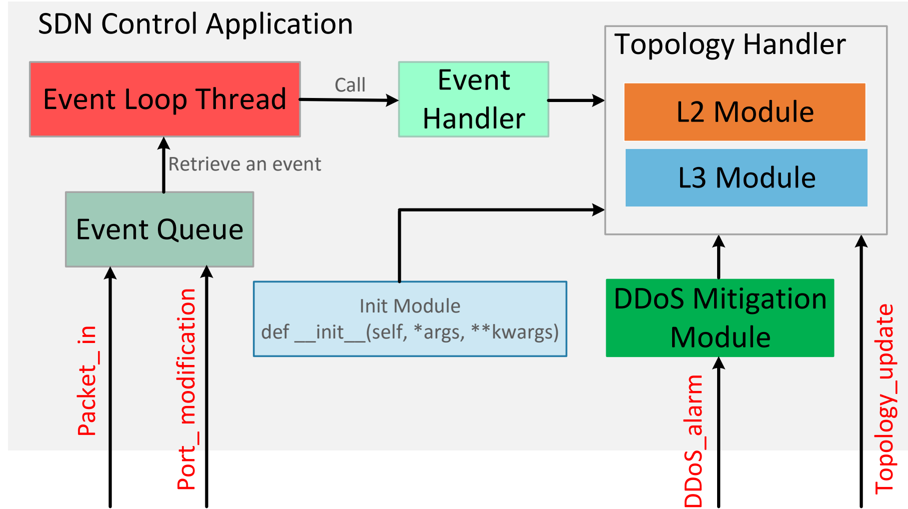


Figure 3.3: *SDN Controller Application Architecture*

L2 and L3 Network Topology View: Shark tank orchestration tool, STO, builds the entire shark tank automatically through VM instantiation and overlay network establishment. We implemented STO such that once the VMs are instantiated and links are established, L2 (connection between hosts and their switch) and L3 topology (connections between switches in private cloud) information are sent to the SDN controller as JSON objects, stored in the topology file of the SDN Controller. RabbitMQ technology [78] is used for sending this information to SDN controller. When the SDN controller application is run for the first time, it reads in the topology, and builds topology data structures accordingly. Now when a switch forwards a packet to the SDN controller for flow rule installation, the SDN controller programs the switch immediately as it already

holds a complete view of the network topology.

Topology Update: When the topology is modified (i.e., a VM joins or leaves the shark tank or a VXLAN link is added/removed) at run-time, the topology information is updated and the controller is notified by STO about the change in the topology (topology update in Fig. 3.3). We have developed a background RabbitMQ service on SDN controller that always listens to topology update notifications so that the SDN controller continuously keeps the latest view of the network topology.

Proactive Flow Installation for DDoS Attack Mitigation: We have developed a DDoS attack mitigation component run via an independent thread that is always listening for DDoS mitigation commands. When SDN controller receives the DDoS mitigation commands, it constructs a match field based on the received DDoS mitigation command. The match field includes the source and destination IP addresses of suspicious traffic. The SDN controller then builds an *action* which includes the mitigation policy (redirecting/blocking the traffic). Our mitigation policy also includes a rule to disguise the IP addresses of the shark tank web server and replace it with that of the original server. This way the suspicious user cannot know that it is actually interacting with the shark tank server (i.e., we do NATing in SDN way). After constructing the match field and action, SDN controller pushes `packet-out` commands to switches so that they forward the suspicious flows according to the action. In addition, the SDN controller modifies the rules on the switches' flow tables through `flow-modification` (proactive flow installation) which include the required actions that the switches have to take when encountering any match in the future.

3.1.2 Experiment

We have performed a set of experiments to evaluate CAAMP on a hybrid cloud. More specifically, the target application that we would like to protect against DDoS threats is hosted on Amazon Web services (AWS) public cloud. The architecture of the target application is depicted in Figure 3.1. The application has as a three-tier cluster using Apache 2.0 as load balancer, an eStore web application in Tomcat 7, and a MySQL database. We use SAVI as the private cloud to host the shark tank. The specification of VMs used in our experiments are jointly presented in Tables 3.1 and 3.2.

We developed a workload generator in Python such that users periodically send HTTP requests to the application to select some items from the database and then wait for a random period of time between 500-950 ms and then send the next request. This way the normal users send approximately 1-2 requests/second. Furthermore, we used Hogna [65] to automatically deploy and monitor the application and act as the autonomic manager in CAAMP architecture. Hogna is a modular framework that follows the MAPE-K loop methodology [73] to monitor applications, analyze data, plan and execute corrective actions to meet application performance criteria. CAAMP takes advantage of Hogna to provide compute elasticity to the application. The elasticity is triggered when CPU utilization goes above 80% or below 40% where in former, an instance will be added as web worker while in latter, one instance will be removed from the original application cluster.

We set up DDoS sensors on both original application cluster and the shark tank to monitor the traffic. The DDoS sensors detect http flood attack which is a denial of service attack where malicious clients try to inundate the resources of applications by sending a large number of requests. DDoS sensors use Snort [79] to detect DDoS threats.

Flavor	VCPU	Memory (GB)	Disk (GB)
Amazon m3.medium	1	3.75	4
Amazon c3.large	2	3.75	32
Amazon c3.xlarge	4	7.5	80
SAVI Small	1	2	20
SAVI Medium	2	4	40

Table 3.1: *Specification of VMs in experiments on Amazon and SAVI clouds*

Table 3.2: *Experiment specification on SAVI cloud*

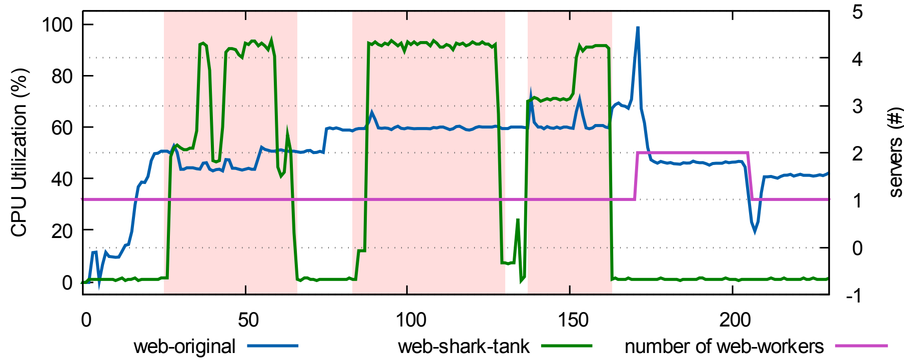
Cloud	No. of VMs	Flavor	Software
Original Application Cluster, Amazon	4-6	Load balancer (c3xlarge), web server(m3 medium), database(c3 large)	Apache Load Balancer, eStore Tomcat Web App, MySQL
Shark Tank, SAVI	2	load balancer (Large), Web server (Medium)	Apache Load balancer,eStore Tomcat Web App
Application Proxy, SAVI	1	Large	Custom Java application
Virtual Switches, SAVI	3	Small	OVS
SDN Controller, SAVI	1	Small/Medium	Ryu Custom Controller Application
STO, SAVI	1	Large	Python Scripts

Once suspicious clients are detected, they are reported to Hogna, shark tank will be created and the suspicious clients are then redirected to the shark tank.

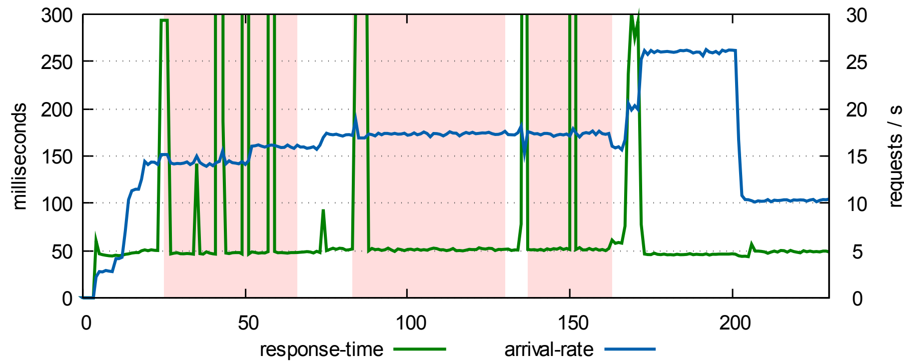
We first demonstrate the DDoS mitigation and elasticity features of CAAMP under various traffic conditions. Second, we show the result of the performance analysis on our SDN controller as one of the core elements of CAAMP.

Before getting into the details of the first result, we explain the main phases of the experiment as follows:

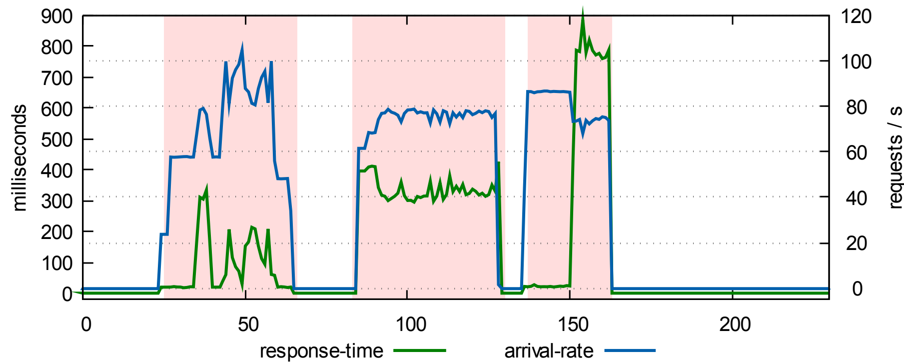
- In the first phase, we introduce multiple attacks to the system and we see how multiple incidents of attack are handled by CAAMP;



(a) *CPU utilization and number of workers.*



(b) *Response time of the web servers in the original cluster.*



(c) *Response time of the web servers in the shark tank.*

Figure 3.4: *DDoS mitigation. Attack periods are shown in shaded background. CAAMP redirects the suspicious traffic to the shark tank on the fly.*

- In the second phase, we introduce one intensive (i.e., large arrival rate) attack;
- In the third phase, we introduce two attacks;
- In the fourth phase, we increase the number of normal users (safe users) to trigger the elasticity feature of CAAMP.

Figure 3.4 presents the result of our experiment with CAAMP in AWS and SAVI hybrid cloud environment. Figure 3.4a shows the CPU utilization of the original application web server (blue line), the shark tank web server (green line), and the number of web workers of the original application (purple line). The purple line shows the elasticity feature of CAAMP where web worker instances are added or removed to meet the required performance metrics. Figure 3.4b shows the arrival rate (blue line) and the response time (green line) of the original web server. The response time uses the left Y-axis while the arrival rate is shown on the right Y-axis. Figure 3.4c presents the arrival rate (blue line) and response time (green line) of the shark tank web server. The X-axis in all 3 graphs shows the experiment iteration number where each iteration takes approximately one minute. The shaded backgrounds in Figure 3.4 indicate the attack period.

In the beginning of the experiment shown in Figure 3.4, we only have normal traffic and the response time of original application server gets stable at around 50 ms. We initiate the first phase of the experiment where from iteration 23 to iteration 60, we start and stop 5 attacks one by one where the attackers send 50 to 100 request/second. Once each attack is started, it increases the response time of the original application server (5 peaks in response time on the second plot). The response time during the attack period goes up to 450 ms in Figure 3.4b, however we did not show the peak response times in order to show the normal response time of original application server

at around 50 ms. When first suspicious traffic is started, it triggers the Snort rule and, as a result, the creation of the shark tank is initiated. When the shark tank is ready, the redirection command will be sent to the SDN controller. The SDN controller programs the gateway to redirect the suspicious client to shark tank. Subsequent attack traffic will be redirected to the shark tank. It can be observed that the CPU utilization of shark tank increases from 0 up to around 90% and the total arrival rate to the shark tank raises up to 115 request/second (see Figure 3.4c). We then stop all the attack traffic at around iteration 60. We can see that the arrival rate and response time of the shark tank decreases accordingly.

We then increase the number of regular users, hence the arrival rate and the CPU utilization of the original web server increases from around 45% to 60% and the response time gets stable again at 50 ms. After some time, In the second phase of the experiment, around iteration 80, we start one attack where the attacker sends 80 request/second. We can see that the attack is detected and redirected to shark tank for further processing. We then do not stop the attack for a while, we can see that the response time and CPU utilization of shark tank stays up. This is an example where attack traffic is kept at shark tank so that attacker receives high response time and assume their attack has been successful and do not adapt their attack strategy. Then we stop the attack, and it can be observed that the arrival rate and response time of the shark tank decrease at around iteration 120 in Figure 3.4c.

In the third phase, at around iteration 132, we start another attack which results in a sudden increase in CPU utilization and response time of original web server. The Snort alert is triggered, and the attacker is redirected to shark tank. We can see how fast SDN controller installs the redirection rules on the gateway where the shark tank

response time and arrival rate start to increase immediately. At iteration 148, we start another attack which is then mitigated and redirected to the shark tank. When the new attack traffic is routed to the shark tank, initially shark tank web server can handle the requests well and the response time is as low as 24 ms from iteration 143 up to iteration 150. But then at iteration 150 due to high arrival rate, the requests start to queue up and we can see the the response time of the shark tank web server increases further to 900 ms and its CPU utilization reaches from 60% to 80% (shown in Figure 3.4a). We then stop the attacks which reduces CPU utilization, arrival rate and response time of shark tank web server.

So far we demonstrated how CAAMP effectively mitigates the DDoS threats by dynamically redirecting the traffic to the shark tank. For testing the elasticity feature of CAAMP, we start the fourth phase where we increase the arrival rate of requests by adding more regular clients. Higher arrival rate raises the CPU utilization of original web server and the autonomic manager triggers the elastic behavior. In Figure 3.4a, we can see that up to around iteration 160 (the point that we increase the arrival rate of normal traffic), the number of web servers in public cloud stays at 1. However, when we increase the arrival rate, we can see in the second plot that the response time increases up to 300 ms and the CPU utilization gets to more than 80%. This situation triggers Hogna's elasticity policy and therefore one more instance is added to the original application cluster to keep the response time low (shown in purple in Figure 3.4a). After one more web worker is added, we can observe that the CPU utilization of original web server gets around 40% and the response time gets improved and stable at round 50 ms again. After a while, around iteration 200, we stop some of the clients which results in the reduction of CPU utilization of the application web server. The CPU utilization gets

decreased below 40% which again triggers the elasticity policy and the extra web worker is removed.

The time to redirect the traffic between the original application and the shark tank is an important metric in CAAMP and it is a function of SDN controller response time and the switch's flow installation time. However, we observed that the redirection time is mainly associated with the response time of the SDN controller. Therefore, we carried out experiments to evaluate the SDN controller performance in processing the mitigation commands.

In order to increase the load on the SDN controller, we set a large number of clients to send traffic (attack and no attack). In doing so, we set up a client overlay network on SAVI separated from the rest of the deployment. We used 8 medium VMs on the client network and on each VM, we setup 40 VXLAN links. Each link is connected to a port with an IPv4 address representing a client (i.e., 320 clients in total). Then, we add routing entries to the Linux kernel routing table for each interface so that each interface (i.e., client) sends/receives traffic independently from other clients on the same VM to avoid ARP flux problem [80]. This way, the SDN controller is responsible for setting up flow rules for each client interacting with the application, in addition to processing the mitigation commands.

We performed two types of experiments by using small and medium size VMs on SAVI, specification of which is defined in Table 3.1, to act as the SDN controller. Figure 3.5 depicts the response time of the small and medium size SDN controller for processing mitigation commands (i.e., proactive flow installation). During this time, the SDN controller is also responsible for installing flow rules for application traffic reaching the switches on SAVI (i.e., reactive flow installation). We performed each

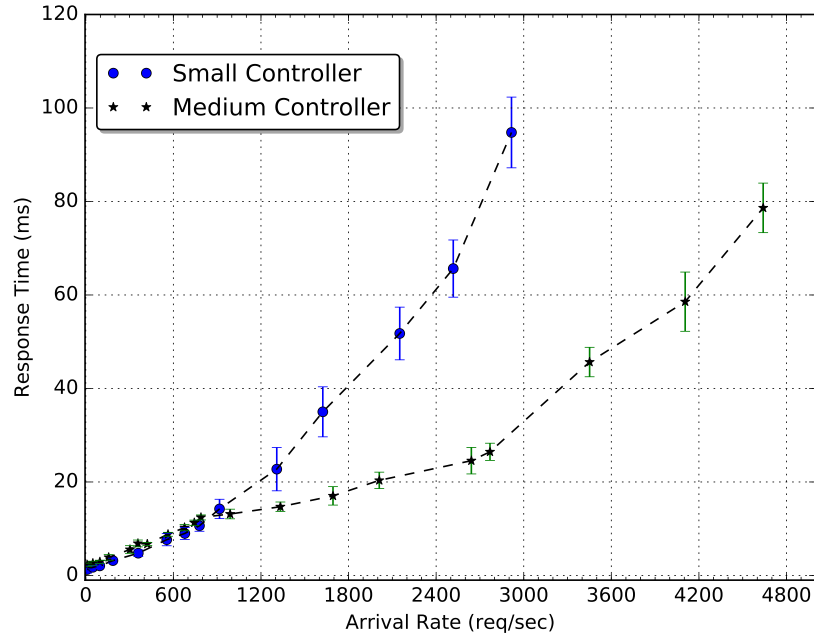


Figure 3.5: *SDN Controller Performance*

experiment type 4 times and calculated the standard deviation as shown in Figure 3.5. The mitigation commands can be any of the following: (1) the redirection of traffic from the original application on AWS to the shark tank on SAVI; (2) the redirection of traffic from the shark tank on SAVI to the original application on AWS; (3) and blocking the traffic in case of real attacks.

As can be seen in Figure 3.5, the response time increases as the mitigation command arrival rates gets increased. For up to 1000 requests/second, we can see that the response time of both controllers are almost the same. Therefore, in terms of performance, for up to 1000 requests/second, a small size controller can handle the requests as good as a medium size controller. We can see that for up to 650 requests/second, the response time for both medium and small controllers is less than 10 ms. However, around arrival rate

of 1200 requests/second, we can observe that the response time of the small controller starts to deviate from that of medium one such that the small controller gets saturated at around 3000 requests/second with response time of 94 ms. In medium controller, however, for up to 2100 requests/second, the response time is under 20 ms. The medium controller finally gets saturated at around 4800 requests/second where the response time reaches 88 ms.

Discussion and limitation: In CAAMP, when suspicious traffic is initially detected, a shark tank is dynamically instantiated on the private cloud, if it does not already exist. While shark tank is being provisioned, suspicious traffic will be reaching the original application. Hence, there is a trade off on the frequency of provisioning/deprovisioning the shark tank; if we aim to quickly mitigate the effect of potential DDoS attacks, shark tank will not be deprovisioned for some time, even though there is no suspicious traffic at the time. Therefore, once created, shark tank will be maintained for some longer time to serve potential incoming suspicious traffic (i.e., deprovisioning once every 6 hours). The amount of time, the shark tank will be maintained for, once created, depends on the likelihood of the application being under attack. On the other hand, if we are more concerned about the resource consumption on private cloud, shark tank will be deprovisioned after some short amount of time if there is no suspicious traffic. As a result, the discussed trade off has to be made by the application owners on the frequency of shark tank decommissioning. Another advantage of CAAMP is that since all the traffic first goes to the private cloud and then to the public cloud, all the redirection between shark tank and the original application happens on the private cloud. As a result, no charge will be applied from the public cloud provider to application owners as all the mitigation process happens before the traffic reaches the public cloud.

CAAMP mainly focuses on the *mitigation* mechanism, therefore, it will be an effective solution to complement detection techniques to reduce large false positives; shark tank provides longer time for attack analysis while it still maintains the service availability until final decision is drawn. Also CAAMP assumes that the attackers are distinguishable via specific attributes such as IP addresses.

3.2 A DevOps Framework for Quality-Driven Self-Protection in Web Software Systems

In Section 3.1, we proposed CAAMP where the adaptation against DDoS attacks is executed at the infrastructure layer by redirecting the suspicious traffic to the shark tank. In this section, we extend CAAMP by containerizing the application to engineer a framework for runtime security adaptation of web software systems that follows the Development and Operations (*DevOps*) approach [81]. Analysis, decision and action components touch upon all phases of the system's lifecycle. The adaptation itself can include actions at the operations side or actions that partially or completely redesign and reconfigure the system. In addition, cost is taken into account for the analysis of security-related incidents. Naturally, this means that the framework allows different experts, including developers, IT staff, QA experts, and security engineers, to work together during the adaptation. The motivation behind a DevOps approach is that security may not be fully and independently guaranteed either during development or during execution. Since the conditions are more than likely to change, it is not always worth the effort to guarantee security at each phase in isolation, as long as we can guarantee it for the overall system during all phases. For this, we will need a channel of

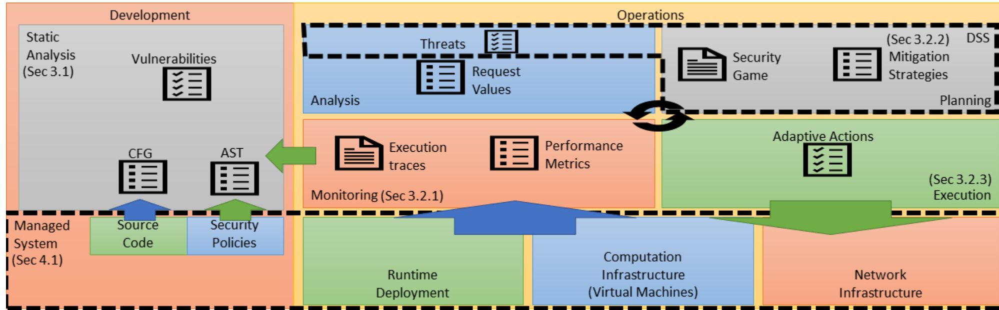


Figure 3.6: *Architecture and modules of the DevOps security framework.*

communication between the different phases (i.e., feedback loops) and a combination of static and dynamic analytical methods.

The operations/runtime component of the framework is based on the MAPE-K architecture for self-adaptive systems, with the following components: (a) **an extended monitoring system**, which, besides resource utilization and performance metrics, also includes meters for security parameters; (b) **a hybrid analysis engine** with a static component to identify security vulnerabilities at development time and a dynamic analysis to correlate static development information with runtime data, identify actual threats and assess their impact; (c) **a decision-support system** as the planner of our MAPE system, which employs game theory and decides upon the appropriate action to defend against threats; and (d) **an execution engine**, which applies planned adaptive actions on the application or its infrastructure.

3.2.1 Framework Architecture and Components

Figure 3.6 shows the overall architecture of the framework. The architecture is split in two phases. First, the *development* phase includes the static analysis, which has access to the static components of the system, like the source code and the security policies. These include the software artifacts of the system, i.e., source code, and accompanying

documentation relevant to security, i.e., the security policies, which can specify concerns like security sensitive operations of the code, access control specifications, traffic history and so on. Second, the *operations* phase includes the dynamic management component of the framework based on the MAPE-K loop. The monitoring module is responsible for gathering metrics about the infrastructure and the software, concerning performance and security. The analysis digests the results from the static analysis, prepares the input for the decision system and evaluates the effects after an adaptive action was taken to verify that security standards were achieved. The decision-support system represents the planning component, which in practice determines the necessary adaptive action. Finally, the execution module implements the decided plan. This module can perform adaptive actions on the infrastructure, network or software part of the system. Between the development and operations phases, there are feedback loops, as per DevOps. For example, the results of the static analysis to reveal potential vulnerabilities are used by the operations at runtime. These results are also used to configure the utility functions of the decision system. When an action is taken, the static analysis runs again on the new system and the management components are reconfigured accordingly.

3.2.1.1 Static Analysis at Development Phase

The static analyzer we use in our system is based on the "Pattern Traversal Flow Analysis" (PTFA) [82]. At this stage, this analysis extracts security models from the abstract syntax tree (AST) and the control flow graph (CFG) of an application.

PTFA computes "definite predicate satisfaction" for each statement in the application and confirms the degree to which the application is secure with respect to security policies. It is straightforward to verify the compliance of predicate satisfaction analysis. If a change

in the application occurs, e.g., a new version or patch is released, the PTFA is executed again on the new version to verify policy compliance. Security protection differences can be efficiently identified by comparing policy satisfaction differences between PTFA models corresponding to different versions of a system. This allows the automatic validation and verification with respect to adaptive actions for security.

3.2.1.2 Operations Phase

Monitoring: in addition to monitoring the performance metrics, we monitor logs for security purposes, including execution traces on the software level and traffic monitoring on the network level. Moreover, the components of the proposed MAPE system track a history of the data they use or produce, like utility functions or adaptive actions taken, to support the adaptive nature of the framework and its online learning capabilities.

Decision-Support System: receiving the analysis of the incoming requests, the DSS [83] aims at selecting the most appropriate countermeasure while considering the satisfaction of the quality goals and the cost of countermeasure. It employs an extensive two-player zero-sum game in which one player is the web application and the other player is the suspicious user. The payoffs for the web application are computed with the aid of utility functions. Utility values are calculated by incorporating the satisfaction of security goals along with the cost of mitigation. Fusing the satisfaction of stakeholders' goals (security policies) and the cost of the possible mitigation strategies at the application level is proposed in [84] by modeling a Normal form Bayesian game. Here, we have defined an extensive form game in which the analysis of the incoming request is received and based on this information the DSS looks for the dominant strategy out of possible mitigation strategies. Employing game-theory in DSS has the advantage of finding the

equilibrium that balances the gain from achieving security goals with the loss incurred by mitigating the attack.

Execute: the execution component answers the questions what to adapt and how to adapt to mitigate security vulnerabilities. Adaptation actions include automatic and complex actions such as redirecting suspicious traffic [85], or issuing a secure patch.

To implement the adaptive actions, we use similar architecture as CAAMP. More specifically, there is a software switch as well as a SDN controller through which the application components are connected on top of an overlay network. When a new patch is issued or when certain requests are to be dropped, a number of commands will be sent to the SDN controller so that the appropriate flow rules are programmed on the software switch. In case of issuing a patch, in addition to the dynamic redirection rule, SDN controller installs rules on the switch to masquerade the patched server IP and change it to the original server IP when responses are being sent to the users. This way the users view of the web server's IP remains unchanged.

3.2.2 DevOps Integration: A Quality-Driven Workflow

The proposed framework is designed to address two main challenges: uncertainty in security and quality tradeoffs. The first challenge refers to the uncertainty around security during software design and execution. Uncertainty comes from evolving attacks, something which cannot be predicted at design time, and changing operating conditions. Investing resources on addressing some security vulnerabilities at development time may prove futile, either because at runtime a new vulnerability may appear under an unforeseen usage scenario or because we failed to properly rank vulnerabilities and we addressed the wrong one. This is an obvious reason why we need a combination of

static analysis during development time and dynamic analysis during execution. The second challenge comes precisely when analyzing the system's security at execution time. It is expected that addressing a vulnerability, while it improves security, it may also affect other non-functional requirements [86, 87]. Any security event, an attack or a fix, may reduce the performance of the system, e.g., increase its response time due to additional checks, or restrict certain functionalities to certain groups of users, and it will most definitely impact the business side of the system negatively, positively, or both. Therefore, we also need coordinated actions to simultaneously achieve or at least guarantee the various goals of the system.

The two challenges motivate the adoption of a DevOps approach to implement self-protection in software systems. On one hand, the approach promotes the integration between development and operations, which in the proposed framework satisfies the need for combined static and dynamic analyses. At first, the developers build the system according to safety standards and policies provided by the security engineers and then test it against known malicious activity (i.e., penetration testing). At this stage, we have a set of known vulnerabilities for our system. With the exception of some easy-to-fix issues, it may still not be appropriate to fix all vulnerabilities due to uncertainty and unexpected costs. The set of vulnerabilities is transferred to the operations team, who deploys the system, releases it to the users and closely monitor the activity around the vulnerabilities. At runtime, the operations team can start to formulate a picture about the "popularity" of certain vulnerabilities, estimate their probability of being exploited by attackers and assess the impact on the rest of the system.

Performance and cost metrics are combined with the runtime monitoring information (including suspicious traffic around the vulnerabilities) to determine (a) the tradeoff

between security adaptation, quality and cost/value, (b) what is the best action according to these tradeoffs, and (c) when is the best time to implement this action. This process is reflected in the analysis and planning component of the proposed framework.

3.2.3 Experiment

To better demonstrate the function and the usefulness of the proposed framework, we built a prototype and put it through a scenario to be used as a first proof of concept. In this section, we describe this scenario, the application that is to be managed and how we implemented the modules of the framework.

Application: the application to be managed in the prototype scenario is a 3-tier application, which allows users to perform operations on a database; read, insert and update data. Especially for reading, data is returned encrypted to the users. The application defines four roles with respect to access controls; administrator, moderator, registered user and undefined (anonymous). In practice, we assume that the users have previously acquired a token, which specifies their role and which they attach to the request. According to the security policies of the application, the administrator has full access to all functions of the application, the moderator may perform database actions (insert and update), but may not request data encryption, the registered user can only read data and an anonymous user does not have access to anything. The application is deployed with Docker on Amazon Web Services (AWS) cloud and includes a scalable application service and a database service. Originally the application is *developed* not to check the token of each request with respect to the access rights and the functions. In practice, all users are treated as administrators and have access to all functions of the application. This can result in a series of potential vulnerabilities, which indeed

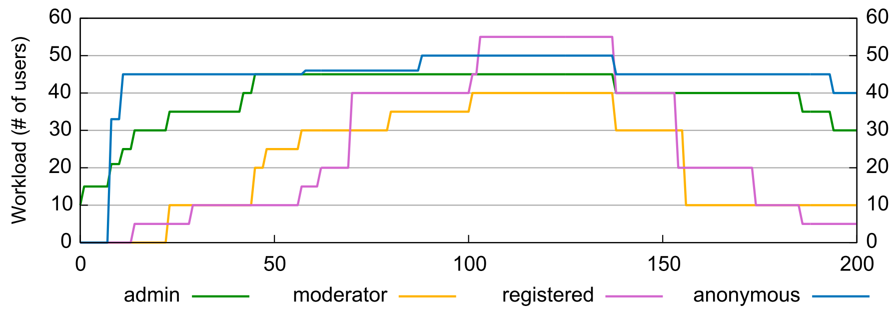
are confirmed by running the static analysis against the software. However, not all vulnerabilities may materialize into actual threats, and even when they do, they may not have the same severity. For example, a moderator asking for encryption is less serious than a registered user asking to write to the database and both are much less serious than an anonymous user asking for encryption. Building on this, the static analysis crosschecks the source code and the policy to identify the vulnerabilities and produces a severity gradient for the various threats: `mod-encryption` $<$ `reg-writeDB` $<$ `reg-encryption` $<$ `anon-readDB` $<$ `anon-writeDB` $<$ `anon-encryption`. Given that a patch to fix the vulnerabilities may be costly and affect the performance, one may risk to have the application deployed like this and react only when and if the vulnerabilities turn into threats.

Once the application is deployed, it passes to the *operations* phase and it is being managed by our framework. At first, the DSS considers the ranking of the vulnerabilities from the static analysis and the security log from the monitoring to set up its game. If an illegal action occurs according to the ranking, the DSS will take one of three actions (do nothing, drop the request, deploy the patch). Utility functions for each action are different. Concerning dropping the request, we consider that requests from different users carry specific values. For example, administrator requests are worth more than moderator requests, which are worth more than those by registered users and so on.

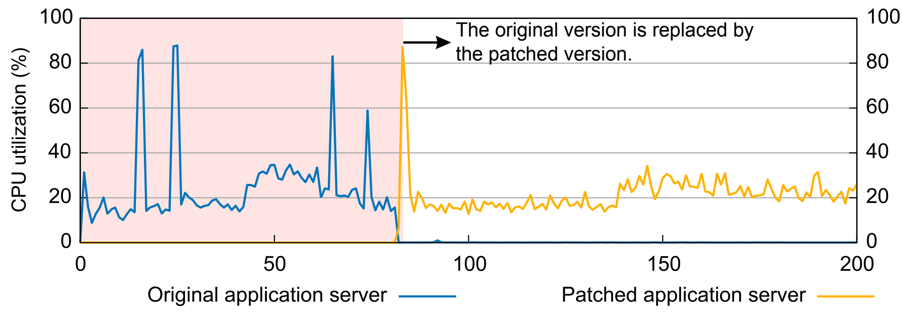
In practice, we can liberally drop anonymous requests, but we should be careful concerning the others. Additionally, when a request is dropped for a particular role the cost to drop another request of the same type increases, to avoid dropping a significant number of requests. Consequently, when a request is dropped the utility functions are appropriately changed for the next decision. With respect to the patch, we assume a

fixed cost to implement the changes and deploy the new secure version. As it becomes obvious, when requests continue to be dropped, at some point the cumulative cost becomes too high and exceeds that of the patch. In this case, the latter action will be preferred and the vulnerabilities will be fixed.

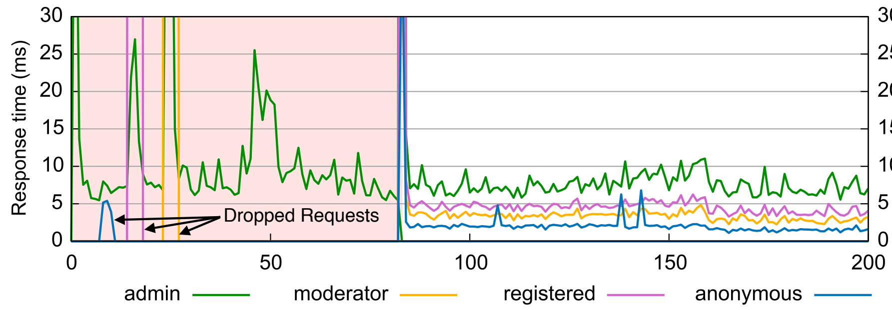
In the experiment to test our prototype, we used a special workload mix with four parts, as shown in Figure 3.7a, to show exactly this behavior of the DSS. Figure 3.7 shows the performance metrics of the system. The first part of the workload contains requests from administrators and anonymous users. This workload does not cause any problems even in the case of the completely unprotected version, since admins have full access and it costs nothing to drop anonymous requests. In the second part, we inject a few illegal requests but of low severity (e.g., `mod-encryption, reg-writeDB`). This causes the DSS to drop some, more expensive, requests. In the third part, we include a large number of anonymous requests and other illegal requests of higher cost. At some point, we observe the scenario, where the cumulative cost of dropped requests exceeds that of the patch and, the DSS decides to deploy the new version. In the charts of Figures 3.7, this event is shown by the difference in the shade of the background. After this action, the static analysis runs again to confirm that the vulnerabilities were fixed and identify new ones if there are any, and the DSS is updated for the new system. The fourth part of the workload includes a full and random mix of requests and roles to show that the new version is robust and secure and can accept all kinds of requests without any security threats.



(a) *Workload*

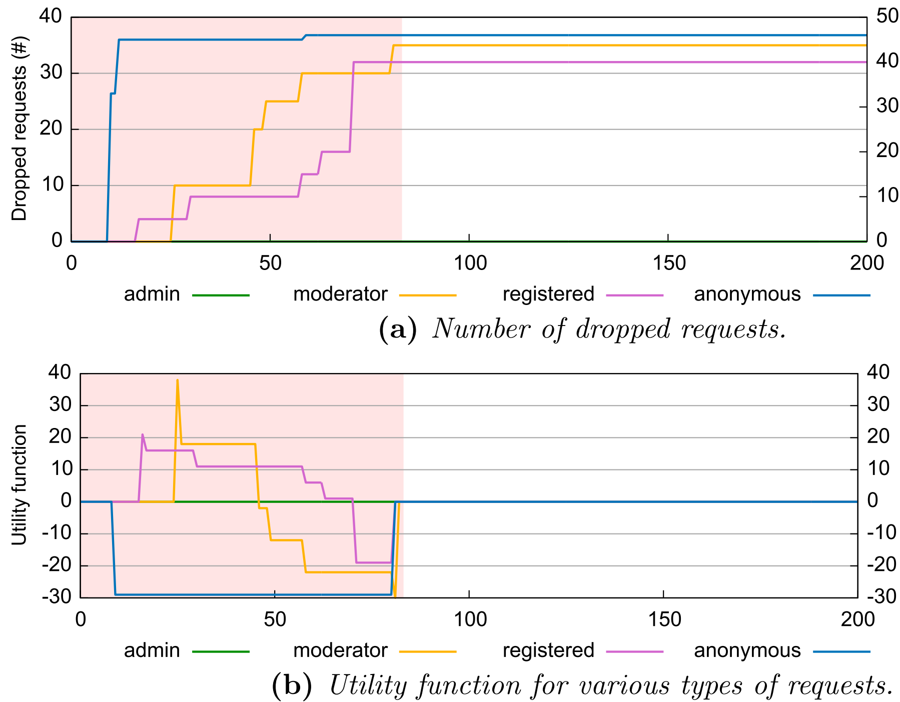


(b) *CPU utilization of the original application and the patched servers.*



(c) *Response times*

Figure 3.7: *Performance results of prototype implementation on AWS.*



(a) *Number of dropped requests.*

(b) *Utility function for various types of requests.*

Figure 3.8: *DSS utility metrics during the experiment.*

3.2.4 Limitations and Assumptions

The focus of this work is on the integration of the individual components (analysis, DSS and execution engine) in a single platform with the purpose of runtime management of the system's security aspects. As such, the implementation of our prototype makes several simplification assumptions with respect to the individual components. Starting with the static analysis, the prototype version is currently based only on the structural identification of vulnerabilities. In reality, it is also necessary to examine data flow as well as control flow. In the scenario we discussed for the prototype, vulnerabilities come from inefficiencies in the design and implementation of the system, but not necessary from malice on behalf of the users. The decision-support system can use online learning and probabilistic thinking to make assumptions for user behaviors and recognize potential

malicious situation, which are promptly incorporated in the game. With respect to mitigation strategies, in the prototype, we only considered two, one in the network level and another in the application layer. Naturally, there can be many more adaptation actions on all layers, each of them with different consequences to the system and the execution engine.

3.3 Summary

In this chapter, we presented CAAMP, a software defined mitigation platform that dynamically mitigates DDoS threats to applications on public cloud using private cloud. We demonstrated the features of CAAMP through extensive experiments on AWS and SAVI clouds.

Our results showed that CAAMP effectively mitigates the severe effects of DDoS attacks. In addition, CAAMP yields elasticity feature to applications on cloud to maintain the desirable performance. CAAMP provides the same functionality to suspicious clients until final decision is made. This way, on one hand, the original application is protected against damages of DDoS threats, and on the other hand, it lowers the rate of false positive alarms which results in customer dissatisfaction and decline in service revenue. As future work, we plan to investigate the end-to-end performance of CAAMP. We are also going to include an in-house detection mechanism in CAAMP where the SDN controller uses network statistics to detect suspicious/malicious traffic.

Inspired by CAAMP, we offer a DevOps solution where for any extension of the platform, multiple stakeholders (developers, QA experts, security engineers) collaborate. The result of this collaboration is improved communication, which can inevitably lead to more efficient decisions both at design and at runtime. More specifically, the collaboration

along with the explicit feedback loops between development and operations will greatly facilitate the runtime management and adaptation of the subject system.

We focused on vulnerabilities around security policies and privileges. Nevertheless, our framework is designed to be extendable in order to accommodate more types of vulnerabilities, attacks and adaptive actions. To achieve this, one has to set up the appropriate monitoring sensors in the system, provide the accompanying alerts in the analysis, update the utility functions with the quality or cost parameters of interest in the planning and eventually specify the adaptive actions in the execution. Our intention is to further facilitate the users of the framework by providing a more formal and structure method (e.g., configuration files or a graphical interface) to specify these details in the framework's components in order to extend it. Another future direction will focus on the adopted DevOps approach. Our goal is to capture and understand the implicit and explicit interactions between the system stakeholders when adapting for security and formally specify them in workflows. This will help to initially guide and coordinate the necessary actions of the DevOps team, but eventually transfer them to tools in order to increase automation and achieve continuous security and self-protection for the system.

Chapter 4

Self-managing Applications with Search-based Network Adaptation

The most common approach to maintain SLA at high load or partial failure is to leverage the cloud elasticity that provides on demand provisioning of computing/storage resources [8]. With mature virtualization technologies in computing, applications can easily adjust their computing/storage demands on the cloud. However, adding more computing resources translates into higher cost for application providers. Moreover adding/removing computing nodes is not an easy task for all applications; for example NoSQL datastores as well as big data analytic platforms such as Apache Spark[1] can not be scaled without experiencing non-negligible overhead associated with data replication, maintaining consistency or job re-scheduling [88]. More specifically in down scaling scenario, the *decommissioning* process of resources might take a long time which renders the whole downsizing ineffective [89, 90]. Other run-time software adaptation such as load balancing are used to dynamically manage the load. However, the load balancer

[1] http://spark.apache.org

itself can become a bottleneck in heavy traffic. This would lead to higher cost and complexity because the load balancer should be eventually scaled.

With advancements in network virtualizations, applications can leverage overlay networks without being locked in to any specific cloud provider. Then using SDN, the flow of application requests through a service can be controlled dynamically based on application policies and requirements. An overlay network on top of cloud provider network brings about more flexibility to application managers to have a fine granular control over their flows.

Hence, in this chapter, we propose a design, implementation, and an algorithm for an application autonomic manager that leverages network virtualization and SDN to dynamically control the bandwidth of application flows to meet the SLAs of its scenarios. The idea of using bandwidth provisioning to improve SLA is not new; however, in this work, we are controlling the bandwidth at application scenario level. In a nutshell, we strive to respond to the following research question:

Research Question: *is it possible for a distributed application in cloud to autonomously use bandwidth control mechanisms for imposing delay on selected flows to maintain SLA without scaling out?*

While, generally, the intuitive strategy to resolve overload or bottleneck problem on the cloud is to increase the amount of resources including bandwidth, it has been shown that reducing the bandwidth may solve the overload problem in some scenarios [91, 92].

Therefore, in this chapter, we investigate the idea of regulating SLAs through imposing delay on some flows within applications overlay networks. To this end, we propose an adaptive management mechanism for applications in cloud that takes advantage of

network programmability and network virtualization to improve the overall performance while reducing the cost and footprint of applications [93, 94]. We deploy overlay networks and virtual endpoints on different components of the application. The network controller programs the virtual end points on each application node according to the application policies. The management mechanism uses a heuristic to select flows of some scenarios that can tolerate delay to improve the response time of other scenarios.

Thus, we present an end-to-end architecture of the management mechanism and implement it on a hybrid cloud. Through extensive experiments, we illustrate that our mechanism improves the overall application performance without imposing extra cost to application owners. Therefore, we answer the research question with the following contributions:

- We propose a fine granular bandwidth control mechanism for cloud applications that improves the application performance without increasing the cost for applications owners.
- We implement and evaluate our cloud-agnostic mechanism in a hybrid cloud setting where extensive experiments are carried out to show the feasibility and advantages of our bandwidth management methodology.

The remainder of the chapter is organized in these sections: in Section 4.1, we present the end-to-end architecture of the management mechanism and explain the overall strategy through an algorithm. Section 4.2 presents the implementation. Section 4.3 shows the results of our implementation on real clouds. And finally, Section 4.4 summarizes the chapter.

4.1 Search-based Network Adaptation

Although computing/storage adaptations (i.e., scaling in/out VMs and containers) have been shown to meet application requirements, they are not always the best solution; from application owners point of view, adding more resources translates into higher cost. It is also challenging for a number of applications such as Big Data and NoSQL data stores to scale out/in due to various reasons including data inconsistency, job rescheduling etc. [88, 89, 90].

Therefore, in this chapter, we propose a different approach to meet application performance requirements by imposing delay on some of the scenarios to meet SLA response time of other scenarios within the application overlay network. Since service-oriented applications rely heavily on communication between different services (and nodes), manipulating the bandwidth within application topology seems a viable solution to maintain service level agreement. Overall, the idea has the following explanation: by delaying some of the application scenarios, we shorten the queues to some congested resources and allow other scenarios to pass faster through the queues. More specifically, we propose to delay flows of scenarios that have response times well below their SLAs. To implement the idea, we design and implement a management mechanism that uses overlay networks and SDN to dynamically throttle some scenarios aiming to improve the response times of other scenarios whose SLA is near violation. Our management mechanism is able to postpone adding extra computing resources to the application as long as possible. The proposed solution is cloud agnostics and applications can manage their network independent of cloud provider network because the application is deployed on an overlay network on top of the cloud provider network.

Figure 4.1 shows the high level architecture of our management mechanism. The

autonomic manager follows the monitor-analyze-plan-execute-knowledge (MAPE-K) loop [73] to mange the application (i.e., *managed system*). The managed system consists of the application along with its virtual links and virtual endpoints. The actions include *bandwidth adaptation* and *adding/removing* VMs.

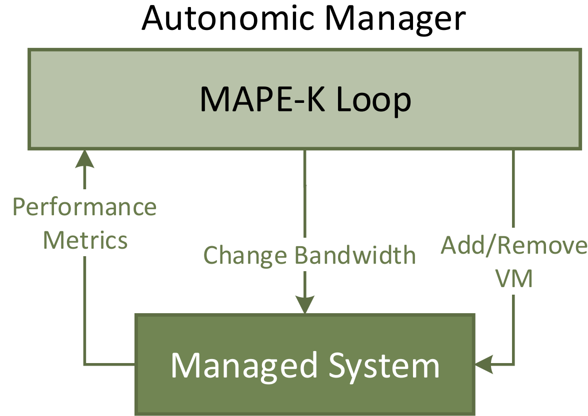


Figure 4.1: *Application autonomic manager and the managed system; autonomic manager manages the bandwidth and VMs.*

In our solution, the autonomic manager automatically builds the application topology; it first instantiate the application nodes and then deploys an overlay network that connects the application components through creating virtual tunneling endpoints (VTEPs) and virtual interfaces (vNIC) on the nodes. After establishing the overlay network, an SDN controller programs the overlay network on the fly based on application autonomic manager policies. Details of the application setup and overlay network establishment were previously discussed in details in Chapter 3, Section 3.1.1. The management mechanism can impose delay on any link within the overlay network to improve the response time of the scenarios whose SLA is near violation. In order to achieve this objective, each application component should be able to distinguish different scenarios. Therefore, before flows leave the egress interface of the application nodes,

they are classified into different classes of scenarios as shown in Figure 4.2; any node of the application that is to apply bandwidth policies should classify the flows based on scenarios. The classification can be implemented on any number of application nodes. Network controller can then program the virtual ports on each node implementing classification to delay certain flows.

Classifying on all the application nodes has its own pros and cons; it provides higher control over flows between any two adjacent nodes in the application topology. On the other hand, it introduces more overhead. So a trade off has to be made between the level of control and the overhead associated with the classification. Figure 4.2 illustrates how the classification can happen at any node of the application. Depending on the method used to apply bandwidth policies, a declassification component might also be needed not to interfere with application layer processing.

4.1.1 Autonomic Manager

The autonomic manger continuously monitors the response time of application scenarios and checks if the SLA of any scenario is about to be violated. If that is the case, actions will be planned to handle the situation that are basically as follows:

1. The autonomic manager first tries to correct the response time by following a greedy hill climbing heuristic to manipulate the bandwidth of application flows.

2. If the bandwith control was not successful, the application is scaled out to prevent SLA violations.

Hill climbing heuristic tries to maximize (or minimize) a target function. At each iteration, it adjusts a single element and determines whether the change improves the

value of the target function. Similarly, when the response time of a scenario is near violation, our management mechanism at every iteration searches among application flows to find a flow that meets certain criteria. If it can find such a flow, it slows the bandwidth of the chosen flow one step down with the goal of improving the response time of a scenario whose SLA is near violation. If no such flow is found, it checks if flows of this scenario has been delayed previously. If yes, it increases the bandwidth of that flow one step up and checks if this action resolves the problem. The heuristic repeats these actions until it exhausts all its options. If the problem is still not resolved, the management mechanism adds extra resources to prevent SLA violations. The management mechanism is summarized in Algorithm 4.1. Before getting into the details of the algorithm, let us define some thresholds that are used in the heuristic based on which the autonomic manager defines the actions:

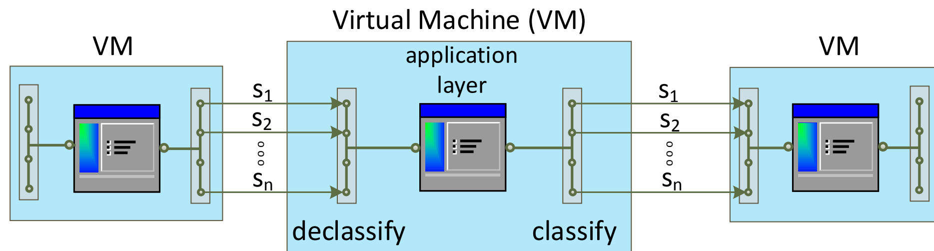


Figure 4.2: *Classification and declassification of scenarios based on the management policy; flows are classified into scenarios and receive a specific bandwidth.*

- **SLA threshold**: it defines the maximum legitimate scenario response time;
- **Trigger threshold**: the goal of the management mechanism is to maintain the response time of each scenario below its trigger threshold. When response time of a scenario reaches its trigger threshold, it triggers our bandwidth adaptation mechanism.

Algorithm 4.1: Adaptation Algorithm:

input : $\mathcal{S}$—vector of scenarios.
input : $\mathcal{R}$—vector of average response times for scenarios.
input : SLA—vector of response times SLA for scenarios.
input : *trigger*—vector of scenario response time thresholds that trigger bandwidth adaptation
input : *candidacy*—vector of scenario response time thresholds based on which a scenario is selected to slow down
input : CPU^{lo}—low CPU utilization threshold.
input : CPU—average CPU utilization of web servers.
input : $\mathcal{H}$—vector of heats for scenarios, where h_s is the heat for scenario $s \in \mathcal{S}$.
input : *heat* — a control number for removing VM
input : n, N — the number of consecutive violations required to trigger an adaptation for bandwidth and VMs respectively.

```
1  foreach scenario s ∈ S do
2      if R_s > trigger_s then
3          if h_s = n then
4              h_s ← 0;
5              f ← {F ∈ S − {s} who meets selection criteria};
6              if f ≠ ∅ then
7                  Decrease bandwidth for f;
8                  return;
9              else
10                 F ← {flows belong to scenario s whose bandwidth can be increased};
11                 if F ≠ ∅ then
12                     Increase bandwidth for one flow in F;
13                     return;
14                 else
                       // bandwidth adaptation exhausted all options
15                     Add VM;
16                     return;
17         else
               // move one step toward bw adaptation
18             h_s ← h_s + 1;
19     else
           // reset any buildup for bw adaptation for this scenario
20         h_s ← 0;
21 if CPU < CPU^lo then
       // cluster underload
22     heat ← heat − 1;
23     if heat = −N then
24         Remove VM;
25         heat ← 0;
26         return;
```

- **Candidacy threshold**: this threshold determines if it is possible to delay a scenario. If the response time of a scenario is above its candidacy threshold, its flows will not be selected to be delayed.

- **Selection criteria**: The flow selected to be delayed has to meet these two conditions: (a) the scenario that this flow belongs to should have response time below the candidacy threshold; (b) the flow should have the highest throughput compared to other flows in the scenario.

In Algorithm 4.1, when the response time of a scenario violates its trigger threshold (line 2), the autonomic manager first performs the hill climbing heuristic with a greedy selection criteria to find a flow to delay (line 5). If such a flow does not exist, the algorithm checks if the scenario whose response time is about to be violated, has been delayed in previous iterations. If it finds such flows within the scenario, it goes one step back and increases their bandwidth one step up one by one (line 12). If the algorithm exhausts all its search options, it adds extra VMs to prevent SLA violations (line 11). In addition, the algorithm continuously checks the CPU utilization of the application. If it reaches below certain threshold, it removes the extra resources (line 16).

4.2 Implementation

We implemented our solution on a hybrid cloud environment. Hybrid cloud environment can be used to provide more flexibility to application owners [85]. Also, we chose a hybrid cloud setting to show our solution is cloud agnostic and does not depend on underlying cloud infrastructure due to using overlay networks. We used AWS as the public cloud and Smart Applications on Virtual Infrastructure (SAVI) cloud [36] as the

private cloud.

Figure 4.3 illustrates the architecture of our adaptive scenario management mechanism on hybrid cloud. The main components of our solution are as follows:

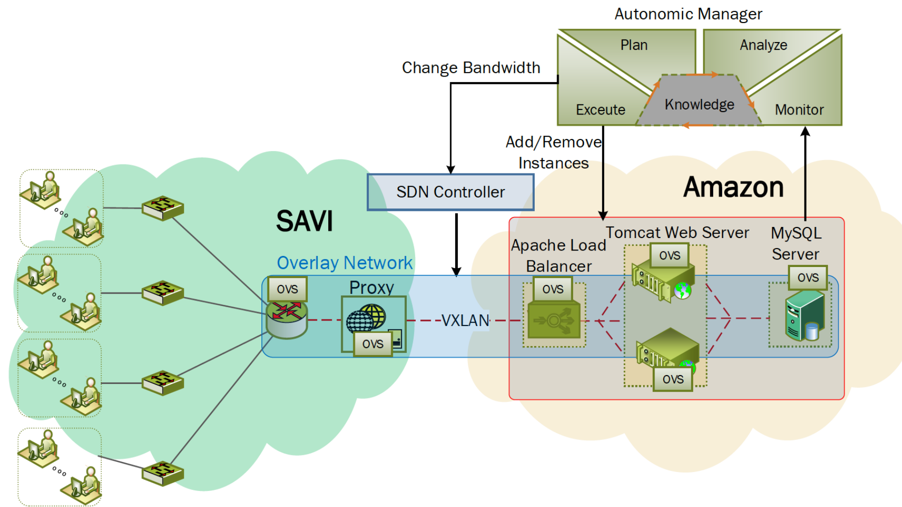


Figure 4.3: *An overlay network that spans over private and public cloud connects application components. The bandwidth rate of any link is programmed on the fly.*

- Application: we use a web application that consists of different scenarios where a scenario uses a chain of services within application topology in a service-oriented architecture. The application represents e-commerce applications where the user sends a request to use any of the offered scenarios, waits for the response, thinks for sometime and then sends the next request. The application is hosted on Amazon EC2 public cloud and consists of a three-tier cluster using Apache 2.0 as load balancer, an eBook store web application in Tomcat 7, and a MySQL database. The application provides four scenarios or use cases:
  - **browse**: the user browses through the book catalog and clicks on various items to see the details;

- **buy**: the user adds a book to the shopping cart;
- **pay**: the user checks out and pays for the content of the shopping cart;
- **auto bundle**: upselling / discounting scenario, where the user receives the opportunity to bundle together related books based on the item currently viewed.

- Autonomic manager: manages the application with regards to application requirements at run-time. It follows a Monitor-Analyze-Plan-Execute(MAPE) loop to achieve application objectives. Basically, the autonomic manager first monitors the response time of application scenarios and checks if the SLA for the response time of any scenario is violated. If yes, it first tries to fix the problem by using bandwidth adaptation. To do so, using a hill climbing heuristics with a greedy criterion, it finds scenarios whose response time is below certain threshold with respect to their SLAs. Among all the candidates, it reduces the bandwidth of the flows belonging to a scenario that has the highest throughput. If no candidate scenario is found, the autonomic manager scales out the application to meet SLAs. Basically, the autonomic manager is responsible to dynamically instantiate and configure the application nodes and links automatically. It performs these tasks with the following steps:

  1. It instantiates the application nodes based on application topology;
  2. It then creates an overlay network that connects application nodes over cloud provider(s) network and configures the nodes accordingly;
  3. After application cluster is ready, autonomic manager iteratively monitors the response time of various scenarios, analyzes and plans actions when the

need arises.

To implement the autonomic manager, we extended Hogna [65] to include network adaption as well. The only adaptation possible in Hogna is scaling in/out the application. The autonomic manager follows MAPE loop that is written partially in Java and Python. XML files are used to describe the application topology including the name of instance images, size, availability zones etc. The autonomic manager uses SNMP and CloudWatch probes on nodes on SAVI and Amazon respectively to monitor application specific metrics. The autonomic manger builds the application cluster dynamically using the topology XML files. Images of application nodes are Ubuntu 14.04 with Open Virtual Switch (OVS) [2] installed. Hence, once the application nodes are instantiated, the autonomic manager creats the overlay network dynamically. It connects to each node and creates a virtual tunneling endpoint as well as an interface and assigns an IP to that interface. Then, based on application topology, it builds VXLAN [95] links to connect nodes to a VM that acts as a switch within the application cluster. The result will be an application cluster that is built on an overlay network on top of networks of SAVI and Amazon.

- SDN controller: programs network flows within the application overlay network based on policies dictated by the autonomic manager. We have implemented a multi-thread control application in Python on top of Ryu [76]. One thread is responsible to respond to requests coming from switches and the other continuously listens for commands arriving from the autonomic manager to apply specified bandwidth rates to interfaces. We use RabbitMQ for communication between autonomic manager and SDN controller.

[2] http://openvswitch.org

- Proxy: to monitor the response times, arrival rates, and throughput of application, autonomic manager makes use of a proxy node to timestamp the requests and responses. The response time will be calculated from the moment the proxy receives the request from the client until the proxy receives the response from the application. Also, in our current architecture, the autonomic manager differentiates various scenarios by using the proxy node. As our solution works on dynamically adjusting bandwidth of scenario flows, we have implemented a method to distinguish flows based on scenarios and then apply the appropriate bandwidth rate on them. Using regular expressions, we have implemented a Java application on proxy that categorizes requests based on scenarios, once requests arrive (i.e., the customer-facing node is the proxy node). Then, since all application flows share common resources, we have implemented the proxy application such that each scenario request will be forwarded to a specific interface. Now that every scenario has its own interface, we can have the SDN controller control the bandwidth rate of that interface according to the adaptation goals. SDN controller uses traffic policing features of OVS to dynamically configure the bandwidth of each scenario. Such categorization and bandwidth management can happen between any two nodes of the application so that resource contention can be controlled in a fine granular manner.

4.3 Experiment

We have assigned users into four groups that are using the application's scenarios. In our implementation, users from each group are behind a NAT gateway and application proxy sees only one IP address per group. We implemented this to be in tune with real

scenarios where users from a household, department etc. use application scenarios. In future, the grouping policy can be used to differentiate between various users (i.e., SLA per user). However, in this work, we do not differentiate between users in a group in terms of SLA. To emulate users, we use a workload generator such that users periodically send requests to use different application scenarios. When they receive the reply, they think for a random period of time (i.e., 500-550 ms as *think time*) and then send the next request. The number of active users in the system changes in range of $[5 \cdots 67]$ during the experiment. Table 4.1 shows the overall distribution of users in four groups. Also Table 4.2 presents the specification of the nodes on hybrid cloud.

Table 4.1: *Distribution of users in groups.*

Group	**G1**	**G2**	**G3**	**G4**
Population	31%	36%	30%	3%

Table 4.2: *Experiment spec on SAVI and Amazon. VMs on SAVI are OpenStack small size.*

Components	**# VMs**	**Flavor**	**Software**
App Cluster (Amazon)	4	Load balancer (c3.xlarge), web server (m3.large), database (m3.medium)	Apache Load Balancer, eStore Tomcat Web App, MySQL
Proxy (SAVI)	1	Small	Custom Java app
Virtual Gateway (SAVI)	1	Small	OVS
SDN Controller (SAVI)	1	Small	Ryu Custom Controller App
Clients (SAVI)	4	Small	Python Scripts

Table 4.3: *Candidacy, Trigger, and SLA thresholds of response time for each application scenario.*

scenario	Candidacy thresh	Trigger thresh	SLA
Buy	158 ms	175 ms	250 ms
Browse	360 ms	400 ms	572 ms
Auto Bundle	540 ms	600 ms	857 ms
Pay	900 ms	1000 ms	1430 ms

Now we show how our bandwidth management mechanism helps the application to maintain its SLA while avoiding adding extra resources as long as possible. We create three major events in which we explore the bandwidth management mechanism. We first explain what we intend to show in each event and then discuss each event in more details.

Event 1: in the beginning of the experiment, we show how by *decreasing* bandwidth of some scenarios, our management mechanism maintains the desired response time for all scenarios.

Event 2: in the second event, we demonstrate how the management mechanism successfully maintains the scenario response times below the desired threshold through *increasing* bandwidth of some flows.

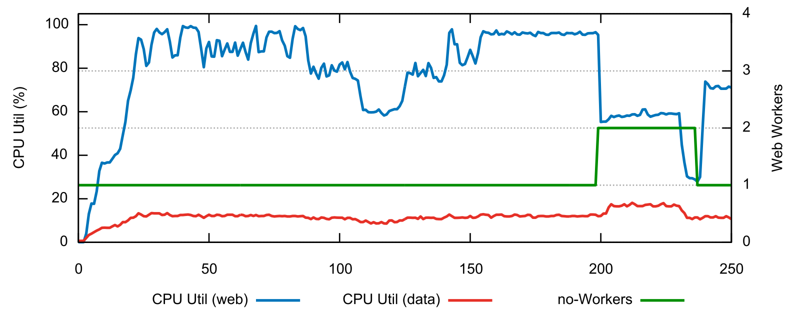
Event 3: in this event, we first show a situation where bandwidth adaptation mechanism exhausts its all options and can no longer improve the response times. In such a situation, the autonomic manager scales out the application and adds a VM to maintain the SLAs. After a while, the workload is decreased and since all response times are well below their SLA, the autonomic manger scales in the application by removing

extra VMs.

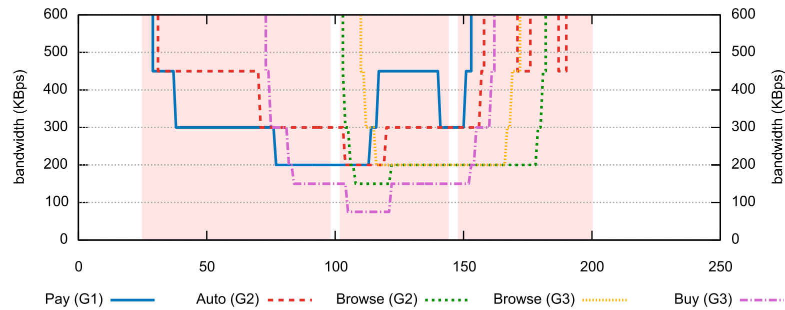
Figure 4.4a shows the CPU utilization of application web worker and database on the left vertical axis and the number of web workers on the right vertical axis. Figure 4.4b shows the bandwidth actions that has been applied to different flows of scenarios. The bandwidth rate has been adjusted in the range of $[75 \cdots 450]$ kbps as higher bandwidth rates did not have any impact on the flows' response time in our application. In Figure 4.4b, we only show the flows whose bandwidth has been changed during the experiment. The three shaded areas in Figure 4.4b represents the three events respectively.

Figure 4.5a shows the scenario response time of *Pay* scenario on the left vertical axis and the arrival rate of requests for the scenario on the right axis. Figure 4.5b depicts the scenario response time and arrival rate of requests for *Auto Bundle* scenario. Figure 4.5c, and 4.5d show the scenario response times and request arrival rates for *Browse* and *Buy* scenarios respectively. The horizontal axis in all 6 figures illustrates the experiment iteration number where autonomic manager collects the monitored data, analyzes and initiates an action based on the monitored data. The duration of each iteration may depend on the frequency of monitoring and it can range from seconds to minutes. In our experiment, each iteration is set to one minute. In all plots in Figure 4.5, we have highlighted three horizontal bands for response time axis. The lower band (i.e., white color) indicates normal response time where no action needs to be taken. The middle band (i.e., dark gray) shows the trigger area in which autonomic manager is triggered to bring the response times back to the white area. And the upper band in Orange is the range at which the SLAs are violated. The autonomic manager should try to keep response times away from this zone. The SLA, trigger and candidacy thresholds for all

scenarios are presented in Table 4.3. In our experiment, we set this trigger threshold to be 70% of the SLA threshold and set candidacy threshold to be 90% of the trigger threshold. Following we explain the various events of the experiments.



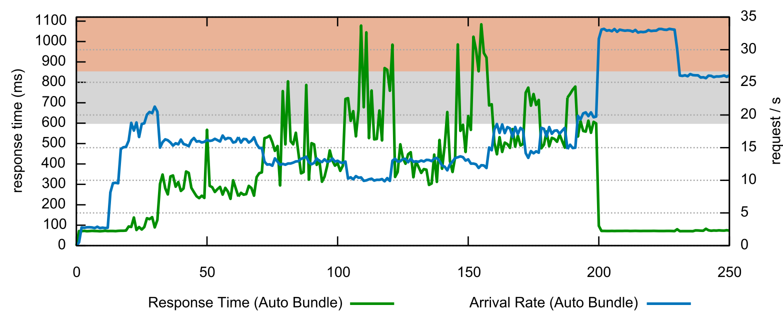
(a) *CPU utilization and number of web workers.*



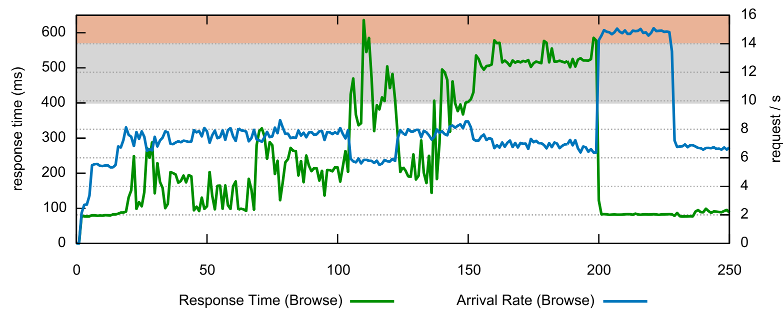
(b) *Bandwidth of scenarios.*

Figure 4.4: *CPU utilization, no of web workers and bandwidth adaptation. Shaded areas represent the bandwidth actions in the three events.*

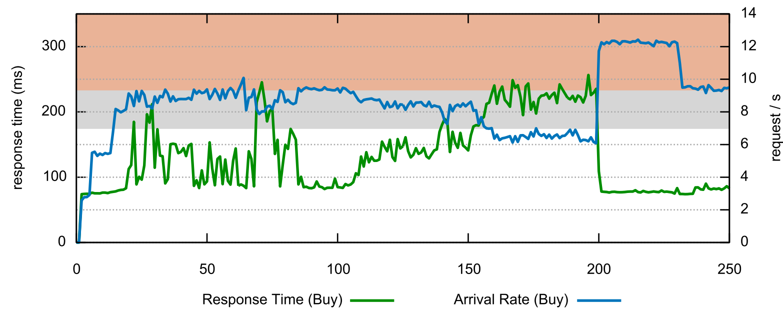
(a) *Pay scenario.*



(b) *Auto Bundle scenario.*



(c) *Browse scenario.*



(d) *Buy scenario (aka add to cart).*

Figure 4.5: *Response time and arrival rate of scenarios during the experiment; the response time axis has been divided into three bands: (a) the white band indicates normal scenario response time (b) gray shows the trigger area and (c) orange area highlights the SLA violation area.*

4.3.1 Event 1

In the beginning of the experiment, as shown in Figures 4.5a, 4.5b, 4.5c, 4.5d, up to iteration 25 all response times of scenarios are below the trigger line and hence no adaptation is performed. At iteration 27, the request arrival rate for *auto bundle* scenario increases (see Figure 4.5b), and, as a result, the response time of *buy* scenario (Figure 4.5d) reaches the trigger line. Our algorithm does not take action right away, instead it waits for two more iterations to make sure the high response time is not a transient effect preventing the ping-pong effect. After the response time of *buy* scenario remains above trigger line for two more iterations, the algorithm starts to perform bandwidth adaptation at iteration 30. From iteration 30 to 32, the algorithm first chooses *pay* scenario to reduce the bandwidth of its flows because it meets the selection criteria. It can be seen in Figure 4.4b, the bandwidth of *pay* scenario goes one step down. After this action, since response time of *buy* scenario has not been corrected yet, at iteration 32, the algorithm performs the second adaptation and reduces bandwidth of *auto bundle* scenario that meets the selection criteria, too. We can see that at iteration 33, the response time of *buy* scenario is corrected and is below its trigger line. In addition, we can see in Figures 4.5a and 4.5b, during these iterations, *pay* and *auto bundle* scenarios are delayed and hence their arrival rates decrease accordingly.

Again at iteration 37, the response time of *Buy* scenario reaches its trigger line and stays the same for 3 consecutive iterations. The algorithm chooses *pay* scenario to reduce its bandwidth that brings the response time of *buy* scenario to the white area. As can be seen in Figure 4.4b, the bandwidth of *pay* scenario flow goes one step down at iteration 39.

By applying the bandwidth management technique, we can see that up to iteration

72, the response times of all scenarios remain below the trigger line and no action is taken up to this point. At iteration 72, the response time of *buy* scenario violates the trigger line, and hence the heuristic chooses *auto bundle* scenario to reduce its bandwidth (see Figure 4.4b). However, this adaptation does not correct the response time of *buy* scenario, therefore another adaptation action is selected which reduces the bandwidth of a flow belonging to *pay* scenario one step down. At iteration 74, the algorithm fixes the response time of *buy* scenario.

Around iteration 78 the response time of *auto bundle* scenario is in trigger range (ie dark gray) so that autonomic manager tries to compensate this by reducing the bandwidth of *buy* scenario by two steps, *pay* scenario by one step and again *buy* scenario by two steps. As can be seen, all above steps make the response time for all scenarios in the normal range (i.e., white area) from iteration 80 to iteration 100.

4.3.2 Event 2

After iteration 100, we can see that the response time of *pay* scenario increases sharply and tends to remain above for a couple of iterations. Therefore, the heuristic tries to fix this by lowering the bandwidth of *buy*, *auto bundle* and *browser* scenarios. Despite such actions, the response time of *pay* scenario still does not get below its trigger line. Hence, the algorithm increases the bandwidth of *pay* scenario two steps up one by one (see Figure 4.4b) at iteration 112 and 115 that fixes the response time of *pay* scenario at around iteration 117 (Figure 4.5a). Meanwhile the response time of *auto bundle* and *browse* scenarios goes above their trigger lines due the delay imposed to them in previous iterations (Figures 4.5b and 4.5c). The algorithm increases their bandwidth one step up at around iteration 122 and 125. We can see that their response times improve and get

below their corresponding trigger lines. Now, at around iteration 125, we can observe that our mechanism successfully maintains the scenarios in good conditions for more than 20 iterations while the CPU utilization remains below 80%.

4.3.3 Event 3

We increased the request arrivals of *buy* and *browse* scenarios at around iteration 140 which then increases the response time of all scenarios. The algorithm tries to correct the response times by managing the bandwidth within the scenarios. It can successfully fix the response time of *pay* and *auto bundle* scenarios by increasing their bandwidth step by step (see iteration 150 in Figure 4.4b). We can see that their response times get below their trigger lines at iteration 155 and 160 respectively. However, the response time of *browse* and *buy* scenarios still remain high despite the algorithm trying to increase their bandwidth (see Figure 4.4b). At iteration 195, the algorithm exhausts all bandwidth adaptations which leads to scaling out by adding one VM. The right axis in Figure 4.4a shows that the number of web workers is increased to 2 at iteration 198. The third shaded area in Figure 4.4b represents Event 3 up to iteration 200. Afterwards, a VM is added and the response time of all scenarios get below their trigger line at iteration 200. We can see that our algorithm delayed adding more VMs as long as possible by managing bandwidth to retain SLAs.

At the end of experiment, the workload is decreased and since the response times of all scenarios are good, the autonomic manager scales in the application by removing one VM while SLAs are still maintained. It can be seen that the number of workers in Figure 4.4a goes down from 2 to 1 at around iteration 240.

4.4 Summary

In this chapter, we presented design, implementation and an algorithm for managing application performance in complex and dynamic cloud environments by dynamic management of network bandwidth. The proposed approach is based on a hill-climbing based bandwidth management that strives to maintain the response time SLAs across all scenarios of an application. This is accomplished by applying flow control policies at the scenario flow level, which is orchestrated by an application autonomic manager within an overlay network. When bandwidth management is exhausted, the autonomic manager provisions new computing resources. We implemented and evaluated the proposed approach on a hybrid cloud environment and showed that the management mechanism is able to successfully meet the application's SLA objectives for a long time without scaling out the compute resources.

Chapter 5

Self-managing Applications with Machine-learning Based Network and Compute Adaptations

A common cloud practice to respond to an increase in application workload is to add more physical resources. However, this practice has high performance overhead and yields high cost for the application owner. A faster and less costly approach is to adjust the soft resources, such as bandwidth. Previously in Chapter 4, we proposed to adapt bandwidth at the application level to meet SLA requirements while avoiding auto-scaling for as long as possible. We used a hill-climbing algorithm to explore bandwidth space to find the appropriate bandwidth rates that satisfy the desired response times. However, this approach can be slow, that is, when the action is actuated, the autonomic manager has to wait to see the effect and then plans and takes the next action. Therefore it is inevitable that in some cases it would take time to come up with a good solution. Since all these decisions and actions happen at run-time, we need to speed up the run-time

reactions. Therefore, to improve the quality of adaptation, in this chapter, we propose a machine-learning model that is able to predict the suitable bandwidth rates that satisfy the SLAs of all scenarios at once.

The autonomic manger controls the bandwidth rates of application scenarios in a fine granular manner when experiencing high load. We build a machine-learning model that is able to accurately predict the appropriate bandwidth rates for application scenarios such that the SLAs of all application scenarios are met. The autonomic manager first analyzes the monitored data, then through use of the model, it plans appropriate bandwidth adaptations. It then executes the actions that are actuated over the links within application cluster via SDN controller. Our solution tries to maintain SLAs for as long as possible without provisioning extra resources. If the autonomic manager cannot maintain the SLAs through smart bandwidth management, it will add new resources to prevent further SLA violations. We implement our solution using overlay networks that are established over the cloud provider network such that autonomic manager is able to manage the bandwidth rates independent of cloud provider. Through comprehensive experiments on AWS, we demonstrate how our adaptive control is able to successfully meet the application requirements. We show that our model captures the dynamics of the system accurately enough such that when its output is applied to the application, it successfully achieves the application objectives. We also compare the new solution with the hill-climbing based one proposed in Chapter 4 and show that our adaptive control yields a faster, more accurate and more efficient adaptation where the number of SLA violations is reduced by 56% [96].

Our contributions in this chapter are summarized as:

- We propose and design a model-based autonomic manager for managing SLAs

of service-oriented cloud applications through smart and fine granular bandwidth management. We show that our model accurately predicts the appropriate bandwidth for all application scenarios given the application metrics.

- We implement and verify our solution through extensive experiments in real cloud settings. We show that our cloud agnostic solution can help applications to meet their service level objectives while avoiding provisioning new resources.

- We compare our model-driven approach with the hill-climbing based bandwidth management mechanism and demonstrate the advantages of the new solution.

Following, we describe the architecture of our adaptive control and explain the MAPE loop within the autonomic manager in Section 5.1. Next, in Section 5.2, we explain the implementation of the solution in the cloud environment. Section 5.3 presents the experiments and results of our implementation. And Section 5.4 summarizes the chapter.

5.1 A Machine-learning-based Autonomic Manager with Fine Granular Bandwidth Control

In Chapter 4, we showed that by dynamically managing the networking flows in a fine-grained way, we can achieve the service level objectives in a soft-manner without the penalties associated with scaling out. Further, since service-oriented applications rely heavily on communication between different services (and nodes), manipulating the bandwidth within application topology seems a viable solution to maintain service level agreements (SLAs). Therefore, following autonomic architecture blueprint [73], we propose an adaptive control (i.e., model-based control) to engineer self-managing

applications such that it autonomously maintains SLAs by programming network flows at run-time in a fine-granular manner instead of scaling out.

Figure 5.1 presents the overall architecture of the solution. Autonomic manager uses a model as knowledge base to plan the adaptions. Following, we describe the model in more details.

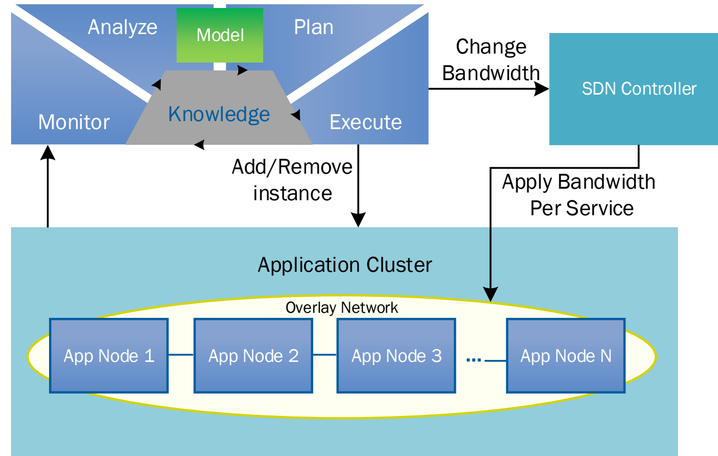


Figure 5.1: *High-level architecture of the adaptive control mechanism.*

5.1.1 Model

We propose a model that presents the relationship between response time, workload and bandwidth rates of the scenarios. To capture this relationship between the response times, workload, and the bandwidth rates, we use a machine learning model formally shown in Equation 5.1. The inputs of the model are the workload vector (W) which includes the number of users per scenario and the vector of scenarios' response times (R). The output of the model is a vector of bandwidth rates of scenarios (BW); that is,

given W, if BW is applied to the scenarios, the response times of scenarios will be R.

We define the effective workload (W_{eff}) according to Equation 5.2 where n is the number of nodes (i.e., virtual machines (VMs)) in the application tier. We only consider application tier VMs as we only scale this tier. Therefore, in order to use the model for when we have only one node in the application tier, we use Equation 5.3. Now BW' is applicable to W_{eff} workload; in order to find BW for W, we use a linear model presented in Equation 5.4. We adopt this formula from our assumptions where we only scale out the application tier with homogeneous VMs and our load balancer uses round-robin policy to equally distribute the workload between the application servers. Therefore, using our method, we only need to collect data for one application setting (i.e., having specific number of nodes in the application tier) and use equations 5.1-5.4 to use the model for varying number of nodes in the application tier.

As part of application performance testing that is done before operation, we tried various values of workload and bandwidth rates. To prevent combinatorial explosion problem, we used the knowledge from application behavior where the workload space to test is limited to the amount that saturates the capacity of that application configuration and the bandwidth rates can be limited between the minimum and maximum possible scenario volumes. It took us three days to collect data for two scenarios. Collecting data and state exploration for more scenarios require a more efficient and faster method which is tackled later in Chapter 6. The goal of the current work is to speed up the run-time reaction, however it has the penalty of slow training process but that happens offline. For the model, we tried a number of different machine learning algorithms including neural network, MLP regressor, SVR, and decision tree regressor. Among all the models, the decision tree model best fits our data set. We use a multiple output decision tree

regression as our model.

$$\overrightarrow{\boldsymbol{BW}} = F(\overrightarrow{\boldsymbol{W}}, \overrightarrow{\boldsymbol{R}}) \tag{5.1}$$

$$\overrightarrow{\boldsymbol{W}}_{\mathit{eff}} = \overrightarrow{\boldsymbol{W}}/n \tag{5.2}$$

$$\overrightarrow{\boldsymbol{BW}}' = F(\overrightarrow{\boldsymbol{W}}_{\mathit{eff}}, \overrightarrow{\boldsymbol{R}}) \tag{5.3}$$

$$\overrightarrow{\boldsymbol{BW}} = \overrightarrow{\boldsymbol{BW}}' * n \tag{5.4}$$

Algorithm 5.1 illustrates the adaptation process of autonomic manager. Autonomic manager uses two criteria to adapt the bandwidth rates which are: (a) the response time of at least one of the scenarios should be below certain threshold with respect to its response time SLA, which we call candidacy threshold (CT); (b) there exists at least one scenario whose response time SLA has not been breached for the past x time units. We call this Time From Last Violation (TFLV). If there exists at least one scenario that meets such criteria, autonomic manger performs bandwidth adaptation (line 3 in Algorithm 4.1), otherwise, it will scale out the application. To prevent oscillation, autonomic manager uses a heat mechanism before acting on a violation or when scaling in the application (lines 2 and 15 respectively).

Algorithm 5.1: Adaptation Algorithm:

input : $\mathcal{S}$—vector of Scenarios.
input : $\mathcal{R}$—vector of response times of scenarios.
input : SLA—vector of response time SLAs of scenarios.
input : $\mathcal{W}$—vector of workload.
input : n—number of application servers.
input : CPU—average CPU utilization of application servers.
input : $heat_o$, $heat_u$—number of consecutive SLA breaches (overload) and underloaded situations respectively.
output : BW—vector of bandwidth rates that satisfies $\mathcal{R}$.
output : S'— a subset of S that meet the bandwidth adaptation criteria .

```
1  foreach Scenario s ∈ S do
2      if R_s > SLA_s for heat_o(s) times then
3          S' ← checkBWCriteria({S − {s}});
4          if S' ≠ ∅ then
5              W ← W/n;
6              BW ← F(W, R) × n;
7              Apply BW to S;
8              heat_o(s) ← 0;
9              return;
10         else
               // bandwidth adaptation cannot fix violations
11             Scale out application tier;
12             n ← n + 1;
13             heat_o(s) ← 0;
14             return;
15 if CPU < CPU^lo for heat_u times then
       // application tier underloaded
16     Remove VM;
17     heat_u ← 0;
18     n ← n − 1;
19     return;
```

5.2 Implementation

We equip our autonomic manager explained in 4.2 with a multiple output decision tree regression model that we developed in Python3 using scikit-learn [97]. The hyper-

parameters set for the model include criterion=mse, splitter=best, max_depth=5, and random_state=0, which attain score function of 0.974.

We performed two sets of experiments on AWS using the setup shown in Figure 5.2; the first one presented in Section 5.3.1 serves as the baseline that illustrates the hill-climbing heuristic previously discussed in Chapter 4; the second one in Section 5.3.2 demonstrates our model-based approach and shows how it successfully overcomes the limitations of the heuristic one. Table 5.1 presents the specification of the nodes on AWS.

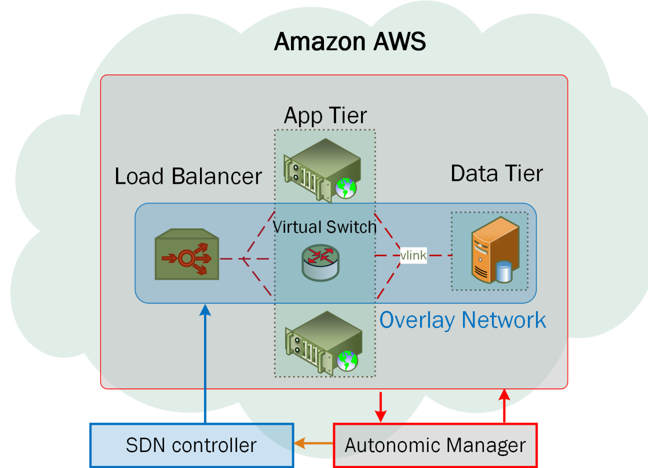


Figure 5.2: *Experiment setup on Amazon AWS.*

Table 5.1: *Experiment specification on Amazon AWS*

Components	# VMs	Flavor	Software
Load balancer	1	m4.large	Apache Load Balancer
Application server(s)	≥ 1	m4.large	book estore using Apache Tomcat
Database server	1	m4.large	mySQL
SDN Controller	1	m3.medium	Ryu
Virtual switch	1	m3.large	OVS

We used the same application previously discussed in Chapter 4. The scenarios

Table 5.2: *Parameters set for the scenarios in all experiments. The autonomic manger is to maintain the* 95th *percentile response time SLAs.*

Scenario	95th SLA	CT	TFLV	$heat_o$	$heat_u$
Browse (Scenario 2)	700 ms	0.9 * 700 ms	1 min	2	
Add to cart (Scenario 1)	40 ms	0.9 * 40 ms	1 min	2	10

considered are: *adding to shopping cart* and *browse*. Each application scenarios has its own response time SLA. Table 5.2 illustrates the parameters set for the scenarios in the experiments. We adapt the bandwidth rates of responses from the load balancer node toward the clients.

5.3 Experiment

We performed the experiments using two scenarios Scenario 1 being *adding to shopping cart* and Scenario 2 *browse the catalog* as we trained our model using these two scenarios.

5.3.1 Experiment 1: Hill-climbing Heuristic

In this experiment, the autonomic manger employs a hill-climbing heuristics introduced in Chapter 4 to maintain SLAs using bandwidth management. Note that the SLA threshold can be set to actual SLA threshold (i.e., hard threshold) or a smaller value than SLA (i.e, soft threshold) depending on the requirements. In the heuristic-based method, the autonomic manger selects scenarios whose response times are below a certain threshold w.r.t their SLAs and has the highest throughput compared to other scenarios. Then it will use a hill-climbing algorithm to continuously reduce the bandwidth of the selected scenario until all scenarios' response times are below their corresponding SLAs. If a bandwidth deduction action results in SLA violation of the selected scenario, the

autonomic manager repeatedly takes one step back to its previous state until it fixes the response time of the selected scenario. If the hill-climbing method cannot fix SLA violations, the autonomic manager scales out the application.

Figure 5.3.a presents the CPU utilization of the load balancer, application servers and database server that can be read from the left vertical axis. The autonomic manager uses detailed monitoring service of Amazon EC2 to monitor the CPU utilization of application nodes where the data is available in 1-minute periods. The horizontal axis shows the experiment iteration number (or the monitoring time) in all graphs. The autonomic manager monitors the application every 20 seconds. The number of application servers are shown at the right vertical axis. Figure 5.3.b shows the workload which corresponds to the number of users in each scenario and Figure 5.3.c illustrates the bandwidth rates of the scenarios. Note that we only show the scenarios whose bandwidth have been adapted during the experiment. Figure 5.4.a shows the response time and arrival rate of Scenario 1 and Figure 5.4.b displays those of Scenario 2 where the response times can be read from the left vertical axis and the arrival rates from the right. The Red dashed lines in both of these graphs depict the 95th percentile response time SLAs. The pink shaded background shows a bandwidth adaptation event and the gray shaded background illustrates an auto-scaling event.

As can be seen in Figure 5.4.a and 5.4.b, up to around iteration 74 all response times are below their SLA and therefore no action is required. However, At around iteration 74 in Figure 5.3.b, the number of users of Scenario 1 increased from 59 to 69 which violates the SLA of scenario 1. Since the response time stays above the threshold for two consecutive iterations, autonomic manager starts off by reducing a proportional amount from the current bandwidth. We found that 15% of current bandwidth would

be a good value w.r.t sizes of scenarios. The autonomic manger reduces the bandwidth rate of scenario 2 in Figure 5.3.c multiple times which finally fixes the response time of Scenario 1 at around iteration 86 in Figure 5.4.a. Since the response time stays above the threshold for two consecutive iterations where autonomic manger selects scenario 2 that meets the selection criteria and reduces its bandwidth.

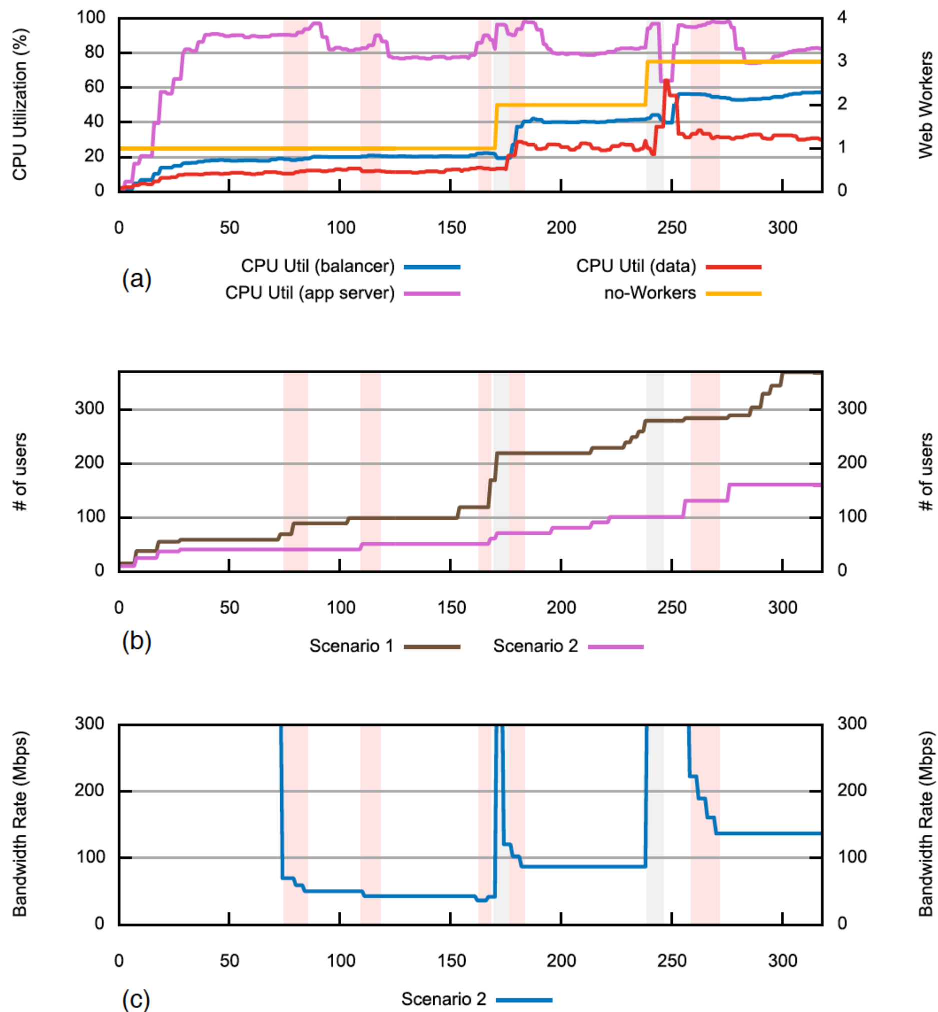


Figure 5.3: *Using the hill-climbing heuristic, bandwidth of Scenario 2 is adapted.*

Other adaptations happen after around iterations 111, 162, and 168 in Figure 5.3.c

which successfully fix the violations. Then at around iteration 169, due to hight number of users, response times of both scenarios go above their SLA thresholds which triggers the auto-scaling policy. A new VM is added to the application tier increasing the number of servers from 1 to 2 at around iteration 172 in Figure 5.3.a. After a new VM is added, the bandwidth rates are reset.

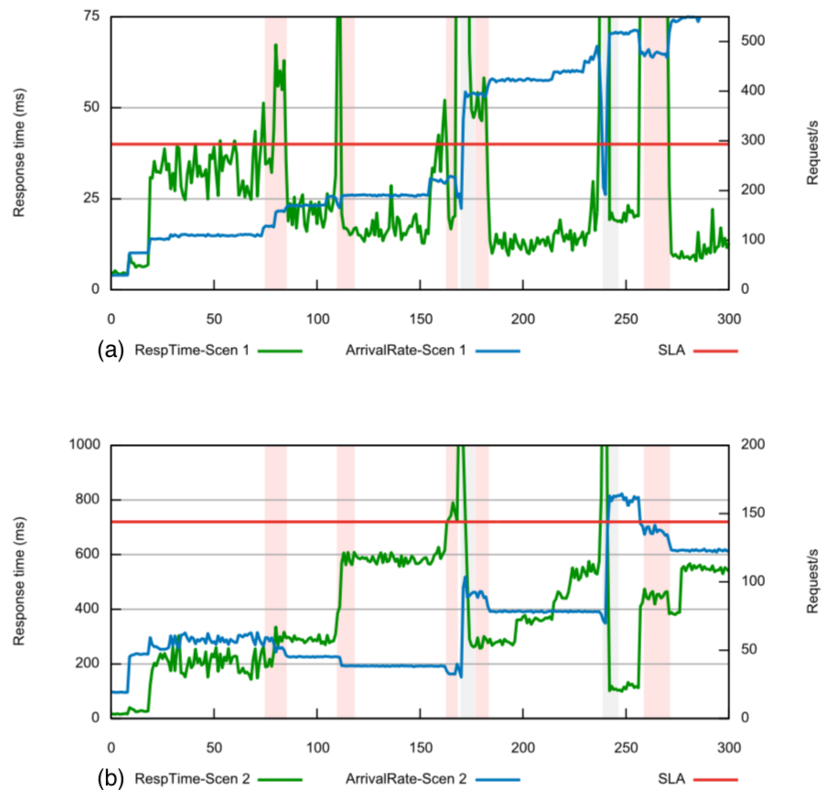


Figure 5.4: *Response time of scenarios in hill-climbing heuristic.*

Even though a new VM is added to the application, the response time of Scenario 1 still tends to stay high which then triggers the bandwidth adaptation. We can see from Figure 5.3.c that the bandwidth rate of scenario 2 is adapted three times until the response time of scenario 1 is fixed at around iteration 184.

Due to large volume of requests, the autonomic manger adds another VM to the application tier to fix the response times at around iteration 240. Then at around iteration 258, the response time of scenario 1 goes over its SLA. We can see in Figure 5.3.c, the bandwidth of scenario 2 is adapted multiple times which finally fixes the response time of scenario 1 at around iteration 272.

5.3.2 Experiment 2: Model-based Adaptation

We keep the experiment settings including the workload and the application configuration exactly the same as previous experiment. In Figure 5.5.b, we can see that as time goes by, the number of users in both scenarios are increasing which increases the response times of both scenarios. Up to around iteration 62 in Figure 5.6.a, the response time of scenario 1 goes a few times above its SLA. However, since the policy is acting on two consecutive iterations, no actions are taken at these times. Around iteration 62, the number of users of scenario 1 increases (see Figure 5.5.b) which causes the response time of Scenario 1 to stays up for 2 iterations above its SLA. Using the model, autonomic manager applies the predicted bandwidth rates to fix the response time of Scenario 1. As can be seen in Figure 5.5.c, the bandwidth rate of scenario 2 is reduced from the default bandwidth to around 69Mbps. We can observe that the response time of scenario 1 gets fixed at around iteration 68. Again at around iteration 74, the bandwidth of scenario 2 is lowered to around 50Mbps that resolves the SLA violation of scenario 1.

We can see from Figure 5.6 that thanks to the model, the autonomic manger is quickly able to find the appropriate bandwidth rates and maintain the application SLAs without scaling out despite the rise in the workload for up to 100 iterations. At around iteration 105, the number of users in Scenario 1 increases further up to 119 users that

breaches SLA of scenario 1. At this time, bandwidth adaptation criteria do not hold (i.e., response time of Scenario 2 is above CT). This means that the current application capacity does not support this workload and bandwidth adaptation cannot be performed. Hence autonomic manager scales out the application to restrain further SLA violations (the number of application workers increases to 2 in Figure 5.5.a.)

When the application is scaled out, the autonomic manger resets the bandwidth rates of all scenarios so that the users can experience faster response time. We can see that scaling out improves the response times of both scenarios. Now let's see if the model is still able to predict the bandwidth rates correctly after this scaling out event. When the number of users in both scenarios is increased from iteration 117 to 124, it breaches SLA of Scenario 1. After the response time of scenario 1 stays up for two back-to-back iterations in Figure 5.6.a. When the model output is actuated over the load balancer interfaces, we can see that the response time of scenario 1 gets adjusted at around iteration 127.

Next, more users are using Scenario 2 which then violates the response time of scenario 1 at around iteration 148. So the bandwidth rate of scenario 2 is reduced to 80Mbps which improves the response time of scenario 1 quickly at around iteration 149. It can be observed from Figure 5.5, that all SLAs are well maintained despite the increase in the workload up to around iteration 189, where both scenario response times gets violated, and the autonomic manger has no choice but to scale out the application. Hence the application is scaled out again (the number of web workers in Figure 5.5.a goes up to 3). However, it takes around 5 iterations for the response times of both scenarios to improve after the new VM is added. This illustrates the benefit of bandwidth adaptation compared to auto-scaling: bandwidth adaptation not only keeps the amount of costly

resources at a lower level, it also performs faster adaptation. At around iteration 212, bandwidth of scenario 2 is adapted which then fixes the response time of Scenario 1 at around iteration 214. The number of users in both scenarios is increased further at around iteration 230 and onward, as we can see the autonomic manger is able to successfully maintain the SLAs of application scenarios which brings us to the end of the experiment.

We just illustrated how our model-based autonomic manager accurately estimated the appropriate bandwidth rates for the scenarios all at once such that the SLA response times of the scenarios are quickly met. We showed that the model predicts the required bandwidth rates even across auto-scaling events.

5.3.3 Analysis of the Results

From the results presented in Sections 5.3.1 and 5.3.2, we can see that the model-driven solution employs an on-line model and can quickly and accurately estimate the appropriate bandwidth rates that resolves SLA violations at once. From the results in the two experiments, we calculated the number of SLA violations of Scenario 1 including the two iterations that the autonomic manger waits to prevent oscillation as well. The model-based autonomic manger reduces the number of SLA violations by 56% compared to the heuristic one. Only the model-based satisfies the 95th percentile SLA while the heuristic one only conforms to the 89th percentile SLA response time. Compared with VM auto-scaling, the time to effect of bandwidth adaptation is much lower. Hence, we can conclude that bandwidth adaptation is an effective and efficient mechanism to adapt the response time of applications and using a model improves the quality of adaptation.

5.4 Summary

In this Chapter, we presented a model-driven application autonomic manager for web applications in cloud utilizing smart bandwidth management within application cluster. Through experiments on AWS, we demonstrated that our adaptive control can quickly maintain SLA response times and reduce the SLA violations by 56% compared to the heuristic-based approach.

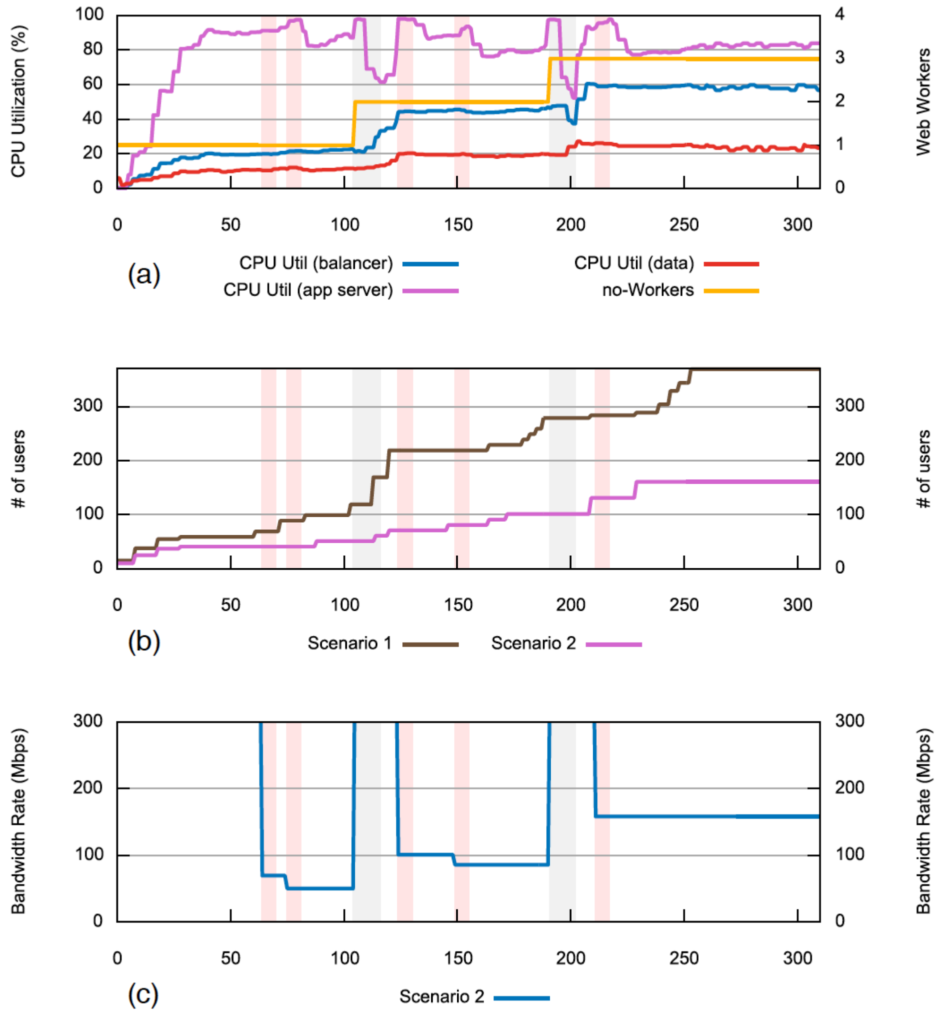


Figure 5.5: *Using the model, bandwidth of Scenario 2 is adapted.*

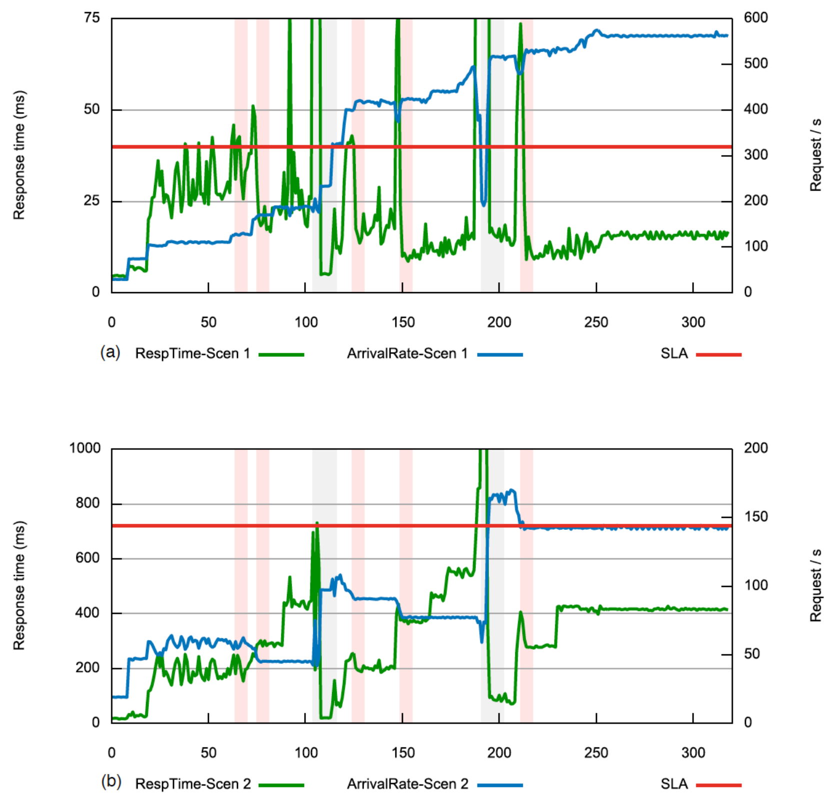


Figure 5.6: *Response time and arrival rates in model-based.*

Chapter 6

Self-optimizing Applications with Scalable Machine-learning based Network and Compute Adaptations

Cloud application owners continuously look for strategies to maximize their profit. They use cloud auto-scaling to adapt to workload changes by dynamically provisioning and de-provisioning resources to match the current demand. However, auto-scaling may not always lead to higher profit due to traffics that generate no or negligible revenue. For example, cloud-based applications are now target of cryptocurrency mining attacks that want to make money using others' compute resources. As a mining malware consumes all available CPU power, the auto-scaler will automatically spawn new instances, allowing the miners to gain huge scalability at the expense of their victims. Recently a company's AWS bill went up from less than $10K to over $100K per month due to illegitimate mining traffic [14].

Also, auto-scaling can become a target of malicious intents where applications are

scaled out due to sudden rise in suspicious and/or illegitimate traffic such as distributed denial of service (DDoS) attacks that waste both the CPU cycles as well as bandwidth of applications on cloud -causing sever economic damage to the application owners [15]. Even when there is no financial or malicious purposes involved, sometimes the application is scaled out as a result of traffic that generate no or negligible revenue such as free-tier scenarios. Therefore it might not be profitable to scale out the application to accommodate such traffic.

In addition, the cost of resources in cloud including CPU, storage and bandwidth is a significant factor that affects the profit. Amazon web services (AWS) company, one of the major cloud providers, also emphasizes the importance of building software using "cost-aware architectures" that manage the cost of using cloud resources [2, 3]. Maintaining performance objectives is another element that impacts the profit where the application owner is obligated to a certain amount of penalty if the service level agreement is violated. Further, due to shared resources, there is dependency between various scenarios of the application where intensity of one scenario affects the response time of other scenarios. This adds another complexity to profit maximization. Hence, maximizing the profit of cloud applications requires an optimal trade-off due to these various, sometimes conflicting factors. To address this problem, we propose a novel application autonomic management system whose goal is to meet application business requirements while optimizing resource allocation using both *compute* and *network* programmability offered by software defined infrastructure. We develop an optimization problem that considers various inputs such as multi-classes of workload and their impact on the revenue, cloud price model, and performance objectives to derive the best course of adaptations that maximizes the profit.

Application autonomic management system continuously monitors the application and solves the optimization problem at run-time to find the most profitable action when needed. Provided that our solution also protects the auto-scalability feature of cloud applications that can be exploited for the reasons previously discussed. Our contributions in this chapter are summarized as below:

- We formally define a profit model that embraces the main factors affecting the profit of web applications hosted in cloud. We develop an optimization problem from the profit model to maximize the profit by using network programmability in addition to compute programmability to better utilize compute and network resources.
- We implement the solution and verify its usability in real cloud environment.
- We compare our solution with a number of baseline approaches. Experiment results demonstrate that the proposed solution improves the profit of the tested application significantly higher compared to the baseline approaches.

Following we present the solution in details in Section 6.1 where we describe the architecture of the solution. We present the decision making process within the autonomic management system in Section 6.2. In Section 6.3, we explain how we implement our solution in real cloud environment. Section 6.4 presents the experiment results. Based on the results of experiments, we compare our solution with some baseline approaches and highlight the advantages of the new approach in Section 6.5. Finally, in Section 6.6, we conclude the chapter.

6.1 Adaptive Load Management of Web Applications on Software Defined Infrastructure

We build an application autonomic management system (AMS) that strives to maximize the profit of web applications on the cloud given two *adaptation knobs*, compute and network. We develop an optimization problem that considers revenue model, price model, and performance objectives to find the best adaptation strategy that maximizes the profit. Our solution addresses two main issues associated with the common auto-scaling approaches: (a) we consider the various classes (or scenarios) of workload and their corresponding revenue, cloud pricing as well as performance goals to perform the most profitable adaptations; (b) we combine network adaptation with compute adaptations to compensate for slow reaction time as well as cost of compute resources in auto-scaling. Consequently, application resources are always allocated such that the overall profit is maximized. We make the following assumptions about the problem at hand:

1. We consider a multi-scenario web application where there is dependency between scenarios due to shared resources;

2. We consider a three-tier application where the application tier is only scaled out/in on demand and use nodes with high capacity in other tiers to prevent saturation;

3. We use homogeneous nodes in the application tier and the requests arriving to the application are equally distributed between application workers.

Figure 6.1 depicts the architecture of our solution. AMS continuously monitors the application and determines if an adaptation is needed. If so, it plans the adaptation by solving the optimization problem and sends requests for adaptation to the *cloud*

controller. Cloud controller receives API calls to add new virtual machines (VMs) or remove existing ones to/from the requesting tenant which will be forwarded to the *compute controller*. It can also receive requests to make changes to the tenant's network which will be sent to the *network controller*. The network controller (i.e., SDN controller) provides an API endpoint to program the requested network resources according to the received API calls. Following, we describe the application and AMS in more details.

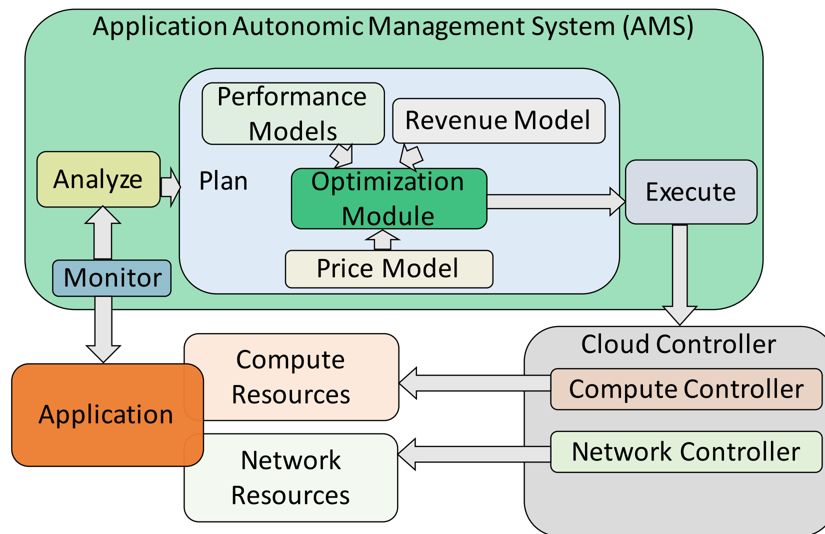


Figure 6.1: *High level architecture of the proposed solution*

6.1.1 Application

We consider an e-commerce web application that has the these scenarios: Browse, Search, Auto-bundle and Pay. Through browsing the customers browse the products on the website. The Search scenario enables the customers to search for a specific item of their choice. Auto-bundle scenario matches the selected items with other items from various vendors and offers them to the customers. And finally, customers use Pay scenario to purchase an item.

In a multi-scenario web application, scenarios have various monetary values which are defined in the revenue model. There are different revenue models for web applications [98, 99]: subscription-based are applications where customers are charged a periodic – daily, monthly or annual– fee to subscribe to a service. Advertising model is another model where the application provider offers free content but advertises other businesses, making indirect revenue. E-commerce and manufacturer applications are other examples of revenue models. A business may combine any of such models to run their web applications in cloud. One of the key factors that affect the profit is the revenue model. Therefore, our optimization module takes into account the revenue generated by each scenario of the web application and decides on the most profitable adaptation. Based on the amount of revenue each of these scenarios generate, we define three revenue models for our web application that mimic a variety of real world web applications which are explained as below:

Revenue Models

We consider three different revenue models: the first one combines advertising revenue model discussed previously with online retailing where the business owner makes money by directly advertising other vendors' products when customers are browsing. From the Search Scenario, the application provider generates revenue by prioritizing products from a specific contract vendor when the customer searches for a specific item. Auto-bundle scenario also generates revenue by recommending bundles of products from their own or other contract vendors. We call this revenue model, *direct advertisement (direct-ad)* model.

The second revenue model that we consider is when there is no advertisement when

browsing and the revenue is generated by offering contract vendors products through search results in addition to revenue generated through Auto-bundle and Pay Scenarios. We call this *indirect advertisement (indirect-ad)* model.

The third revenue model follows a model where the application owner does not advertise or prioritize any specific vendor products when the customers are browsing or searching. Only the Auto-bundle and Pay scenarios generate revenue for the application provider. We call this revenue model *no advertisement (no-ad)* model.

6.2 Models and Run-time Adaptation

In this section, we describe how the planner decides and plans about the adaptations. To this end, we will first discuss the performance models in Section 6.2.1 that are used to estimates performance indicators given a potential adaptation. Then we will present the optimization module in Section 6.2.2 that uses various inputs including the metrics estimated by the performance models to find an adaptation that maximizes the profit.

6.2.1 Performance Models

The performance objective of our AMS is to meet response time SLAs of all the scenarios of the web application. As mentioned before, the control knobs in our solution are auto-scaling and bandwidth rates. To maintain the response time SLA of all scenarios, AMS needs to know the response time of each scenario when bandwidth rates, number of nodes in each tier and the workload are known. Therefore, we develop a model, M, presented in Equation 6.1, that takes the workload, number of application tier VMs and the bandwidth rates of scenarios and will estimate the response time of application

scenarios at time k. Note that in our model, we only consider the nodes in the application tier according to our second assumption.

$$\overrightarrow{rt}_k(\vec{w}_k, \overrightarrow{bw}_k, m_k) = M(\vec{w}_k, \overrightarrow{bw}_k, m_k) \tag{6.1}$$

In equation 6.1, $\overrightarrow{rt}_k$ is a vector that includes the response times of the scenarios at time k where rt_{ki} presents the response time of Scenario i at time k. The workload, $\vec{w}_k$, is a vector that includes the number of requests per each scenario at time k where w_{ki} defines the number of requests in Scenario i at time k. The bandwidth rates of the scenarios at time k are represented by $\overrightarrow{bw}_k$ where the bandwidth rate given to scenario i is defined by bw_{ki}. The number of VMs in the application tier at time k is defined by m_k.

To build M, we tried a number of different supervised machine learning algorithms including neural network MLP regressor, SVR, and decision tree regressor. Among all the models, the decision tree regression best fits our dataset and we use it to build M. Decision trees are non-parametric learning methods that predict values by learning simple decision rules inferred from the data features. In order to train the machine learning model off-line, we need to collect various values for the features including workload, bandwidth rates, and the number of instances in the application tier. To prevent state space explosion of the features during data collection for training, we apply a number of techniques to prune the state space for the inputs of the model which are explained as follows.

First, to prune the state space of the workload, we use domain knowledge expertise that originate from the application under test (AUT). As a part of performance testing procedure, which happens before operation, application and user profiling determine

the approximate distribution of requests. For example, we use following questions to prune the state space for the workload:

- How is the distribution of user requests w.r.t the application scenarios? A possible answer would be on average 40% of users browse and only 3% of users end up buying their selected items (i.e., an application with the hit rate of 3%).
- How the user load is expected to grow with time?
- How long does it take for a specific user action to achieve its peak load?
- For how long the peak load will continue?

Following such performance testing procedure, we define a distribution (D) for the proportions of requests in each scenario at any given time with respect to the number of requests in other scenarios.

Second, in our earlier work [93] discussed in Chapter 4, we applied a hill-climbing heuristic with greedy selection criteria to adapt the bandwidth rates given to application scenarios. This way, we postponed scaling out as much as possible to meet the response time SLA of application scenarios and reduce the cost. The idea is when response time SLA of a specific scenario is violated, the AMS dynamically reduces the bandwidth rate of other scenarios that can tolerate delay in a hill-climbing fashion to fix the SLA violations. Hence, in the new solution, we take advantage of this approach in the data collection phase to prune the state space.

Let us define W to be the maximum number of requests that saturate the application tier when only *bandwidth adaptation* is used to fix the SLA violations using the hill-climbing method. To find W, we perform a number of experiments using hill-climbing approach with having only 1 node in the application tier while increasing the number

of requests aggressively. Although training M can be done using any number of VMs in the application tier, for simplicity, we use 1 VM in the application tier for training. After finding W, using D, we explore the workload space in various experiments using the hill-climbing technique. Now the data collected during these experiments will include various values for the response times, workload, and bandwidth rates that are used to train M.

Since we train M with only 1 node in the application tier, M would be able to estimate response time when the number of nodes in the application tier is 1.

In order to use the machine learning model at run-time where the number of application tier VMs can be more than 1 due to scaling, we adopt Equation 6.2. The reason that we can adopt this formula is because of our third assumption that the requests are equally distributed between application workers. In previous work [96], we showed through experiments that the error of Equation 6.2 is negligible enough. We follow the same approach to build machine learning models that can estimate arrival rates and throughput of the application scenarios given the same inputs. These models are collectively used in our optimization module when a potential adaptation is evaluated for profit maximization.

$$\overrightarrow{rt}_k(\vec{w}_k, \overrightarrow{bw}_k, m_k) = M(\vec{w}_k/m_k, \overrightarrow{bw}_k/m_k, 1) \tag{6.2}$$

6.2.2 Optimization Module

In this section, we explain our optimization module used to determine the adaptation strategy that maximizes the profit. We formally define an optimization problem to maximize the profit of web applications hosted in cloud in Equation 6.3 subject to

Table 6.1: *Symbols used in the optimization problem and their description.*

Symbol	Description
π_k	profit
R_k	revenue
I_k	instance cost
D_k	data transfer cost
P_k	SLA violation penalty
X_{ki}	throughput of scenario i (req/s)
λ_{ki}	request arrival rate of scenario i (req/s)
w_{ki}	number of requests in scenario i
bw_{ki}	bandwidth rate applied to scenario i (kbps)
rt_{ki}	response time of scenario i (msec)
m_k	number of application tier instances
k	monitoring time
Δk	monitoring interval time
r_i	revenue for completing one request of scenario i
S_{req_i}	average size of scenario i requests (KB)
S_{res_i}	average size of scenario i responses (KB)
C_d	cost for transferring 1 KB of data
F_{p_k}	penalty function
τ_i	response time SLA of scenario i (msec)
BW_{min}	lowest possible bandwidth rate (kbps)
BW_{max}	highest available bandwidth rate (kbps)
K	Minimum number of instances at the application tier
L	Maximum number of instances at the application tier

constraints 6.10, 6.11, and 6.12. The inputs to the optimization problem are workload, performance objectives, revenue and price models. The output is the adaptation types and quantities that optimizes the profit.

Whenever an adaptation is to be performed due to response time SLA violations at time k (i.e., monitoring time), the AMS uses the optimization module to perform adaptations that maximizes the profit for the next time interval, Δk. In the objective function presented in Equation 6.3, we look for the optimal bandwidth rates for each scenario and the optimal number of instances at the application tier such that the profit

is maximized.

For an application hosted in public cloud, the profit (π) can be calculated using the revenue generated by fulfilling application requests (R), minus the cost of instance (I), data transfer cost (D) and the penalty imposed by SLA violations (P) as shown in Equation 6.4. The amount of bandwidth rates assigned to each scenario should be between a minimum bandwidth rate (BW_{min}) and a maximum bandwidth rate assigned to the application tenant (BW_{max}) in cloud as shown in Equation 6.10. Also the total bandwidth rates assigned to the scenarios should not exceed BW_{max} (Equation 6.11). The number of instances in the application tier is limited by a maximum (L) and a minimum (K) that are defined based on performance and cost constraints from the application owner. Table 6.1 summarizes the symbols used in our optimization problem along with their description.

$$\max_{\overrightarrow{bw}_k, m_k} \quad \pi_k(\vec{w}, \overrightarrow{bw}_k, m_k) \quad \text{s.t.} \quad 6.10, 6.11, 6.12 \tag{6.3}$$

where

$$\begin{aligned} \pi_k(\vec{w}_k, \overrightarrow{bw}_k, m_k) = R_k(\vec{w}_k, \overrightarrow{bw}_k, m_k) - I_k(m_k) - \\ D_k(\vec{w}_k, \overrightarrow{bw}_k, m_k) - P_k(\vec{w}_k, \overrightarrow{bw}_k, m_k) \end{aligned} \tag{6.4}$$

$$R_k(\vec{w}_k, \overrightarrow{bw}_k, m_k) = \sum_{i=1}^{N} X_{ki}(\vec{w}_k, \overrightarrow{bw}_k, m_k) * \Delta k * r_i \tag{6.5}$$

$$I_k = m_k * C_{vm} * \Delta k \tag{6.6}$$

$$D_k = D_{k_{in}} + D_{k_{out}} = C_d * \Delta k * ((\sum_{i=1}^{N} \lambda_{ki}(\overrightarrow{w}_k, \overrightarrow{bw}_k, m_k) * S_{req_i}) + (\sum_{i=1}^{N} X_{ki}(\overrightarrow{w}_k, \overrightarrow{bw}_k, m_k) * S_{res_i})) \tag{6.7}$$

$$P_k(\overrightarrow{w}_k, \overrightarrow{bw}_k, m_k) = \sum_{i=1}^{N} F_{p_{ki}}(\overrightarrow{w}_k, \overrightarrow{bw}_k, m_k) \tag{6.8}$$

$$F_{p_{ki}}(\overrightarrow{w}_k, \overrightarrow{bw}_k, m_k) = \begin{cases} PR & \text{if } rt_{ki} - \tau_i > 0 \\ 0 & \text{if } rt_{ki} - \tau_i \leq 0 \end{cases} \tag{6.9}$$

In Equation 6.5, revenue (R_k) is calculated at time k by adding the products of X_{ki}, which is the estimated throughput of application per scenario i for the next time interval, Δk, and revenue generated per each completed requests in each scenario, r_i (i.e, \$0.2/req). X_{ki} is the model that estimates the throughput of Scenario i given the workload, bandwidth rates of the scenarios, and number of instances in the application tier after Δk. In Equation 6.6, instance cost (I) is calculated by the product of number of instances at time k (m_k), the cost of instance per time unit (C_{vm}) and Δk. N is the total number of scenarios of the application.

Equation 6.7 calculates the data transfer cost which includes the data that is transfered to the application ($D_{k_{in}}$) as well as the data that is transferred from the application to outside of cloud ($D_{k_{out}}$) during Δk. In equation 6.7, λ_{ki} is the request arrival rate model for scenario i that estimates how many requests from scenario i will arrive in the next time interval if $\overrightarrow{bw}_k$ is applied to scenarios and there are m_k instances at the application tier. The inputs to the arrival rate model are $\overrightarrow{w}_k$, $\overrightarrow{bw}_k$ and m_k. Average size

of scenario i requests and responses in KB are S_{req_i} and S_{res_i} respectively. The cost of data transfer per 1KB is indicated by C_d.

We define the penalty of SLA violations using Equation 6.8. To calculate the penalty, we use a penalty function defined in Equation 6.9 where PR is the penalty rate per SLA violation. According to the penalty function, if the response time exceeds its corresponding SLA, τ_i, the application owner is obligated to the fine rate of PR (i.e, \$0.1 per violation.) The response time model that estimates the response time of scenario i at time k given $\overrightarrow{wk}_k$, $\overrightarrow{bw}_k$ and m_k is indicated by rt_{ki}. As mentioned before, for estimation of arrival rates, throughput and response time models in our optimization problem, we use machine learning models presented in Section 6.2.1. The accuracy of the models are calculated using the coefficient of determination, R^2, which are presented in Table 6.2.

$$\forall i \in [1, N], BW_{min} \leq bw_{ki} \leq BW_{max} \tag{6.10}$$

$$\sum_{i=1}^{i=N} bw_{ki} \leq BW_{max} \tag{6.11}$$

$$K \leq m_k \leq L \tag{6.12}$$

Table 6.2: *Accuracy of machine learning models*

Model	**Accuracy**
Response time	90%
Arrival rate	90.1%
Throughput	90.3%

6.3 Implementation

We implement our solution in a hybrid cloud setting as shown in Figure 6.2. The AMS fetches the monitoring sensors every 20 seconds. AMS analyzes the monitoring data at each time interval by comparing the average response time of each scenario against the scenario response time SLA. Our policy is to take actions after a number of consecutive SLA violations to prevent the "ping-pong" effect [100]. In our experiments, we define the number of violations before taking action to be 3. Further, when average CPU utilization of application tier remains under 40% for more than 20 consecutive intervals, the application is scaled in. If no SLA violation happens for a specific number of intervals, set to 10, and there are some scenarios whose response time can be improved (determined by comparing the current response time to a ratio of the SLA (set to 50%), they are added to a list to increase their bandwidth rates, if there is room. If an action needs to be taken, the analyzer asks the planner for the adaptation type and amount.

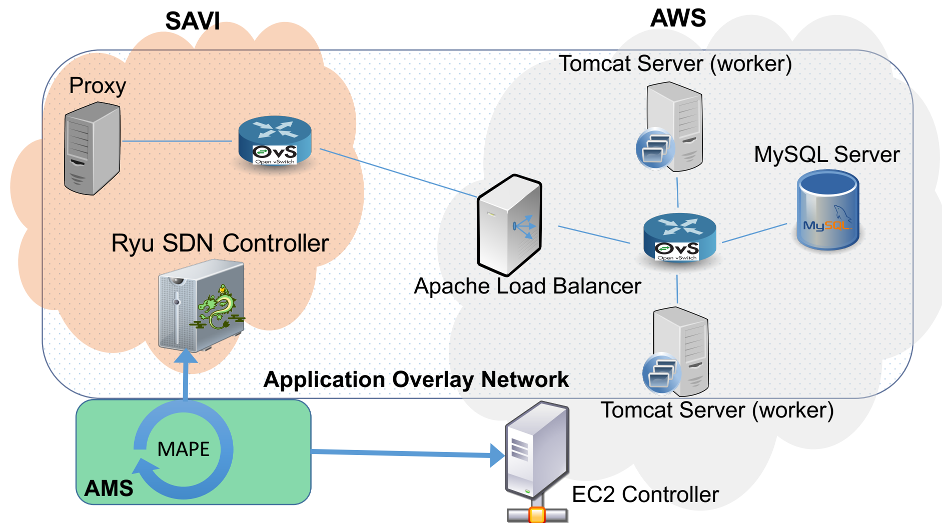


Figure 6.2: *Implementation on hybrid cloud*

The core of the planner is the optimization module that uses a number of online

models to estimate the result of potential adaptations. Then given the workload, performance metrics, SLAs, revenue and price models, it finds an adaptation strategy that maximizes the profit as described in Section 6.2.2. We develop the optimization problem in Python3 using optimization module in Scikit-learn library[97]. We use Sequential Least SQuares Programming (SLSQP) module to implement the profit optimization problem. When there is a need for action, the optimization module will solve the optimization problem at run-time. The solution of the optimization problem may include compute, bandwidth or a combination of both adaptations which will be forwarded to the execution module. Table 6.3 shows the specification of the experiment nodes.

Table 6.3: *Experiment spec on SAVI and AWS. VMs on SAVI are OpenStack small size.*

Components	**# VMs**	**Flavor**	**Software**
App Cluster (Amazon)	On demand	Load balancer (m4.xlarge), web server (m4.large), database (m4.large)	Apache Load Balancer, eStore Tomcat Web App, MySQL
Proxy (SAVI)	1	Small	Custom Java app
Virtual Switch (SAVI)	1	Small	OVS
Virtual Switch (AWS)	1	m4.large	OVS
SDN Controller (SAVI)	1	Small	Ryu Custom Controller App

6.4 Experiment

We perform 5 sets of experiments to compare and contrast our proposed solution with model-based bandwidth approach proposed in Chapter 5 and VM auto-scaling (*auto-scaling-only*). For all the experiments, we use the same workload and the goal is to see how different approaches perform with respect to performance, cost, and profit. Figure 6.3 presents the workload that we use in all of the experiments. The workload

follows a periodic pattern that combines several normally distributed requests resembling workload pattern of an online business application during a day, a month or a year where the number of requests is peaked at certain point of time. Table 6.4 presents the parameters set in our experiments[1].

Table 6.4: *Parameters*

Experiment Parameter	**Value**
C_d	0.0045
C_{vm}	0.0011
PR	0.85
S_{req}	0.1KB
S_{res_1}	25.576KB
S_{res_2}	39KB
S_{res_3}	52.3KB
S_{res_4}	105.8KB
τ_1	1800ms
τ_2	1600ms
τ_3	1000ms
τ_4	1000ms
BW_{min}	$20 * 10^3$kbps
BW_{max}	$32 * 10^4$kbps
K	1
L	10

First, we present the experiment results for *auto-scaling-only* approach in Section 6.4.1 where the only adaptation option is to add or remove VMs dynamically to maintain SLA or reduce the cost respectively. We then present the results of model-based bandwidth approach in Section 6.4.2 where AMS tries to maintain SLA using bandwidth adaptation as the first adaptation choice. If not, it then scales out the application. Later, we equip the planner of AMS to solve the optimization problem presented in Section 6.2.2 where the AMS tries to find the optimized course of adaptations, including auto-scaling and

[1] We use generic unit for prices but we maintain realistic ratio among the prices of different resources.

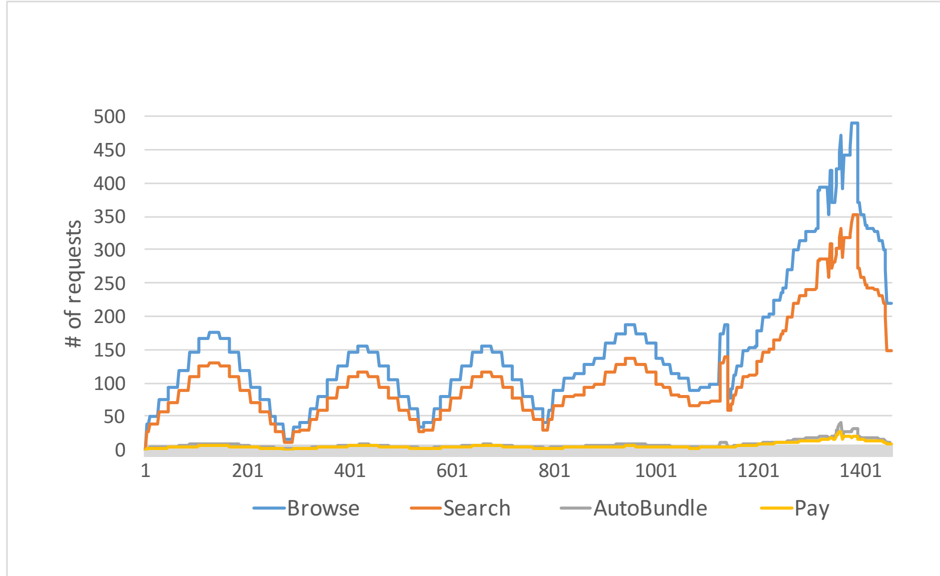


Figure 6.3: *Workload*

bandwidth adaptations to maximize the profit. The experiment results of this approach are presented in Section 6.4.3. In Section 6.5 the results of all experiments are compared and discussed.

6.4.1 Auto-scaling-only Approach

In this section, we present the results of the experiment that we perform using auto scaling as the only adaptation strategy to deal with response time SLA violations according to the state of the art.

Figures 6.4 depicts the application cluster performance metrics including CPU utilization of the three tiers of the application that are shown on the left vertical axis. The number of application tier servers (web workers) that are scaled during the experiment are shown on the right vertical axis. Figures 6.5a and 6.5b illustrate the performance metrics for the scenarios where the average response time of scenarios are drawn in the

left vertical axis and the average request arrival rates of the scenarios are shown on the right vertical axis. In Figures 6.5a and 6.5b, the SLA violation regime is shown in Orange.

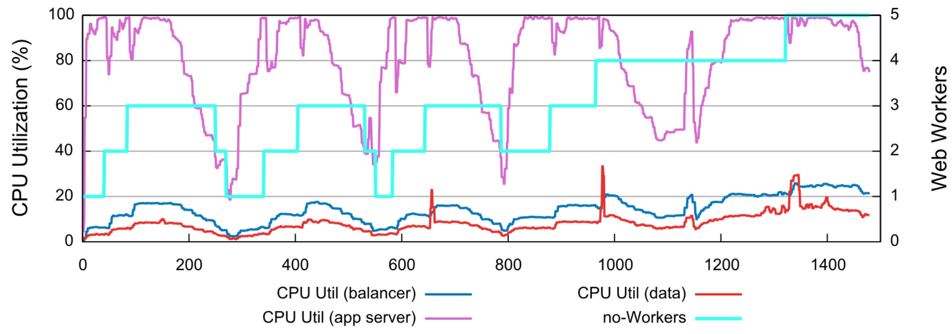
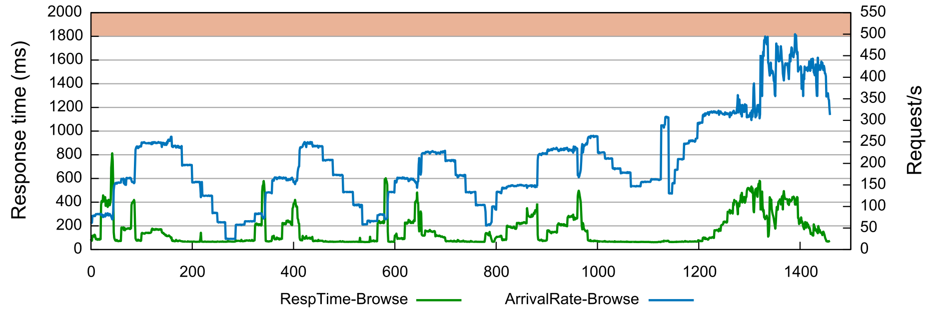
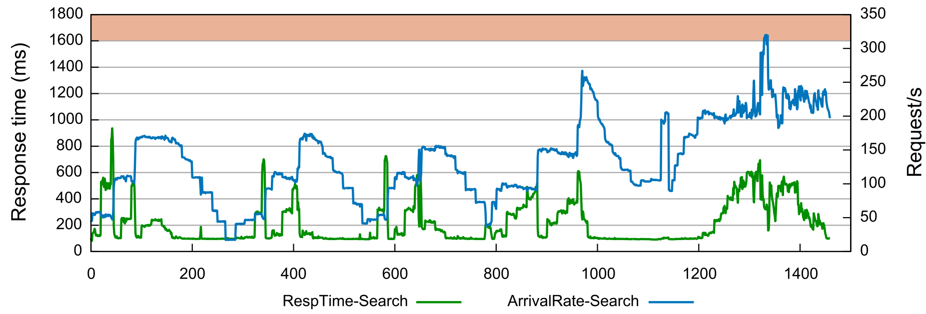
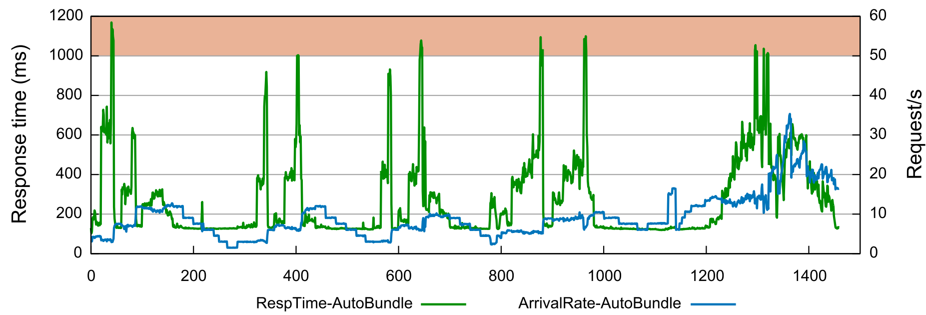


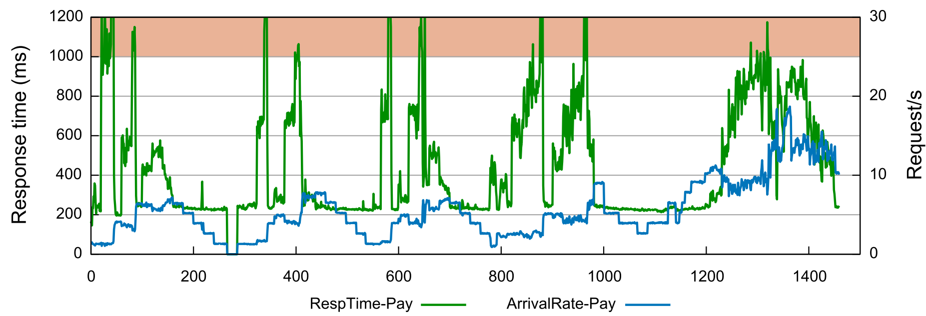
Figure 6.4: ***Auto-scaling-only:*** *CPU Utilization and no. of workers*





(a) *Response time and arrival rates of Browse and Search*





(b) *Response time and arrival rates of Auto-bundle and Pay*

Figure 6.5: ***Auto-scaling-only***

As can be seen in Figures 6.5a and 6.5b, when response time of any of the scenarios exceeds the SLA threshold as many times as the SLA violation limit, AMS scales out the application tier as shown in Figure 6.4 and adds more instances to the application cluster to prevent further SLA violations. When the application is scaled out, the response time violations are fixed. When the condition for scaling in is checked, an instance is removed to reduce the cost. In Figure 6.4, AMS dynamically adds VMs to fix SLA violations and removes VMs to reduce the cost.

6.4.2 Model-based Bandwidth Approach

In this section, we present the results of the experiment when bandwidth adaptation is performed as the first adaptation strategy to meet the SLA requirements and reduce the cost according to [96]. AMS uses a decision tree model to decides how much bandwidth should be reduced from candidate scenarios to fix the SLA breaches. If bandwidth adaption is unable to maintain the performance, the application is then scaled out.

Similar to previous section, Figure 6.6 shows the performance metrics of the application cluster. Figure 6.7 demonstrates the bandwidth rates that AMS apply on the scenarios during the experiment to meet the SLAs. Note that the bandwidth rates of scenarios that have not been selected during the experiment are not shown in the graph not to obscure other values. Figures 6.8a and 6.8b show the metrics of the scenarios throughout the experiment.

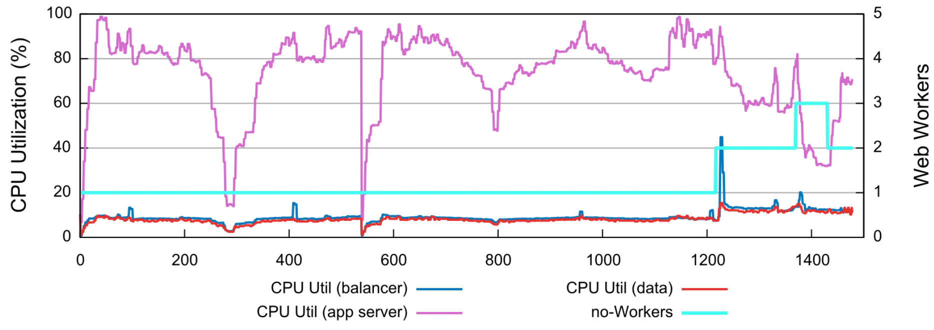


Figure 6.6: ***Model-based bandwidth approach:*** *CPU Utilization and no. of workers*

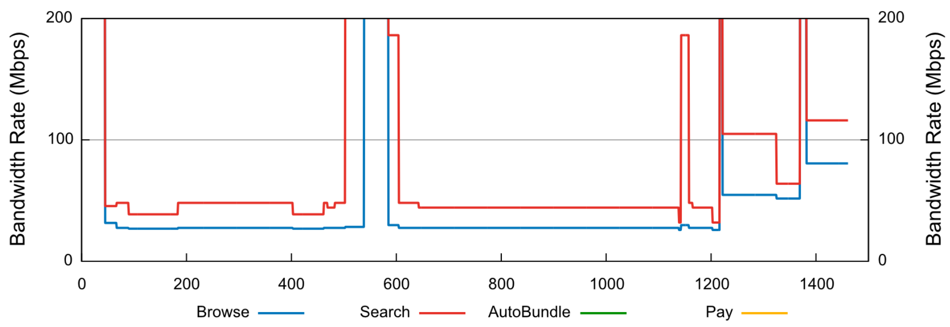
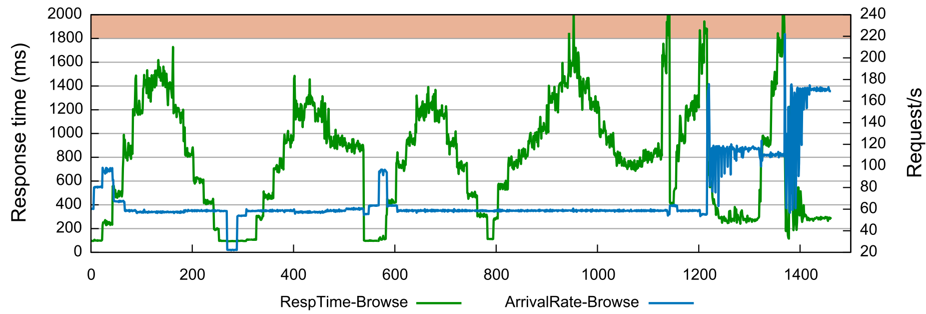
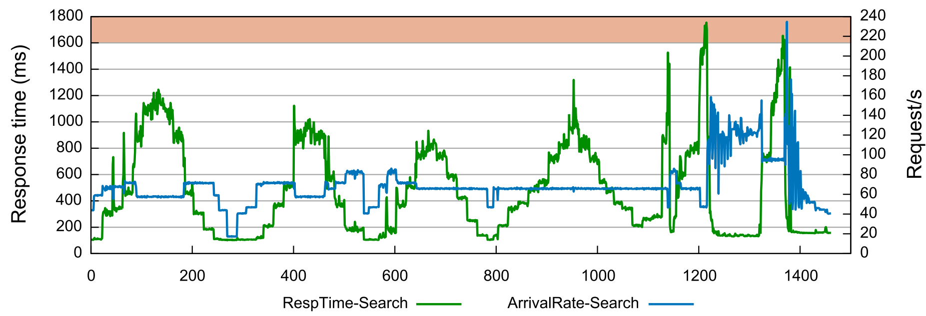
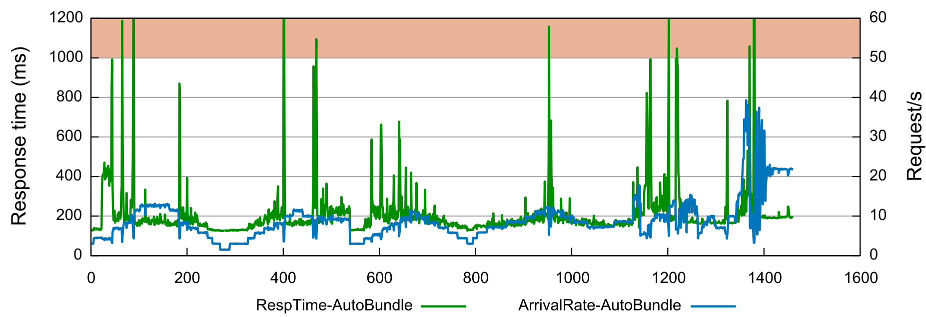


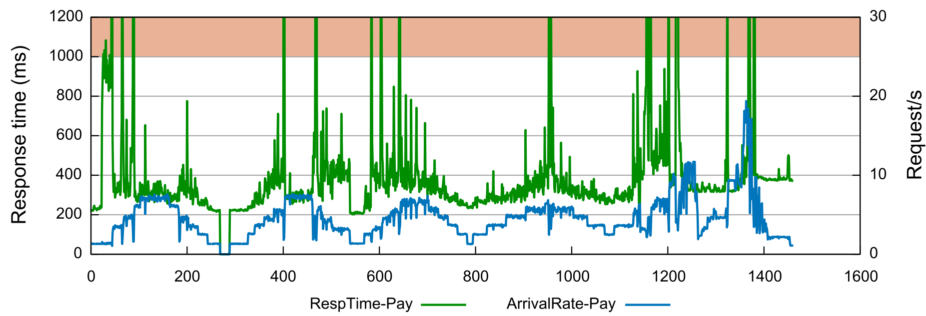
Figure 6.7: ***Model-based bandwidth approach:*** *Bandwidth rates*





(a) *Response time and arrival rates of Browse and Search*





(b) *Response time and arrival rates of Auto-bundle and Pay*

Figure 6.8: ***Model-based bandwidth approach***

It can be seen that this time, application AMS adapts the bandwidth rates of application scenarios to prevent further SLA violations. As seen in Figure 6.7, only bandwidth rates of Scenarios Browse and Search are adapted. It is only at around iteration 1200 in Figure 6.6 where AMS is unable to maintain the performance using bandwidth adaptation. Therefore, it scales out the application tier to maintain the SLAs.

6.4.3 Optimization-based Approach

We implement the optimization problem presented in Section 6.2.2 in Python3 and equip the planner module of AMS with the optimization module. If an action should be taken to maintain the SLAs, AMS solves the optimization problem online, given the performance and economic metrics to find an adaptation strategy that maximizes the profit for the next time interval. The adaptation may include any of the adaptations possible such as auto-scaling and bandwidth adaptations. To represent various business cases, we consider three revenue models (RMs) that were previously discussed in Section 6.1.1. The specifications of the three revenue models are presented in Table 6.5 where $r_1 - r_4$ apply to Scenarios Browse, Search, Auto-bundle and Pay, respectively. For example, the application owner makes \$0.06 by completing one Browse request. We perform 3 experiments using the optimized solution and considering one of the revenue models at a time. The results are presented in Sections 6.4.3.1, 6.4.3.2 and 6.4.3.3 respectively.

Table 6.5: *Three revenue models where $r_1 - r_4$ define price unit (e.g, \$) per request for Browse, Search, Auto-bundle and Pay respectively.*

Revenue model	r_1	r_2	r_3	r_4
$Direct-ad$	0.06	0.3	0.5	0.5
$Indirect-ad$	0	0.2	5	4
$No-ad$	0	0	3	6

6.4.3.1 Optimized adaptation using direct advertisement (Direct-ad) revenue model

Figures 6.9, 6.10, 6.11a, and 6.11b show the results of the optimized management approach using *direct-ad* revenue model.

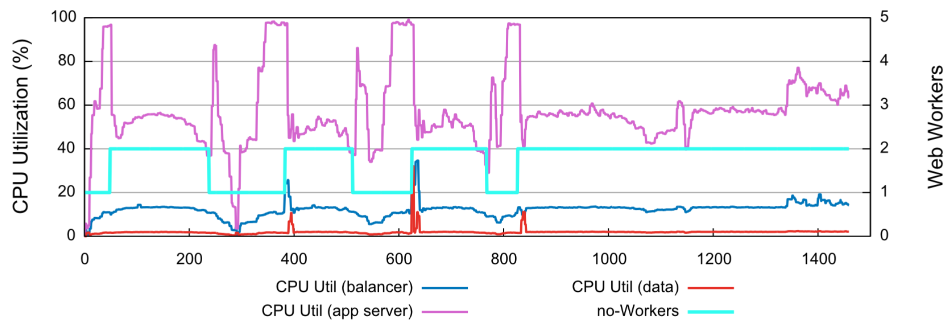


Figure 6.9: ***Direct-ad:*** *CPU Utilization and no. of workers*

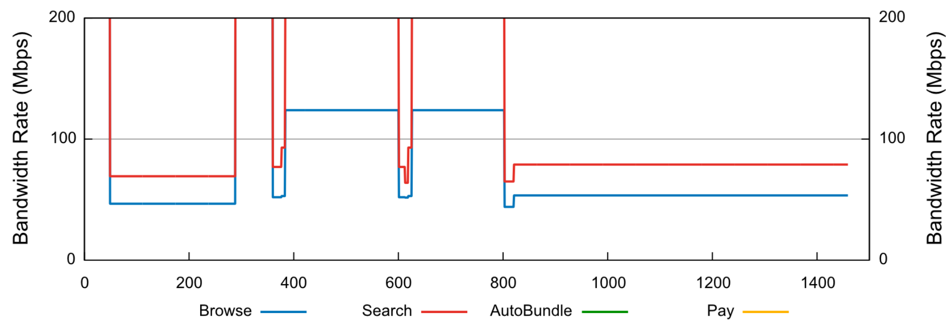
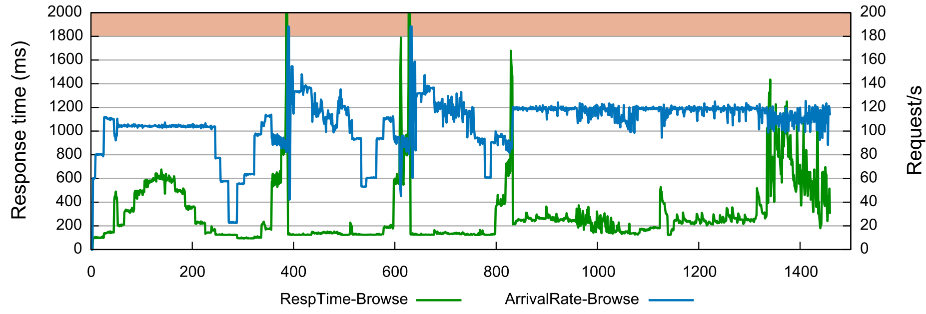
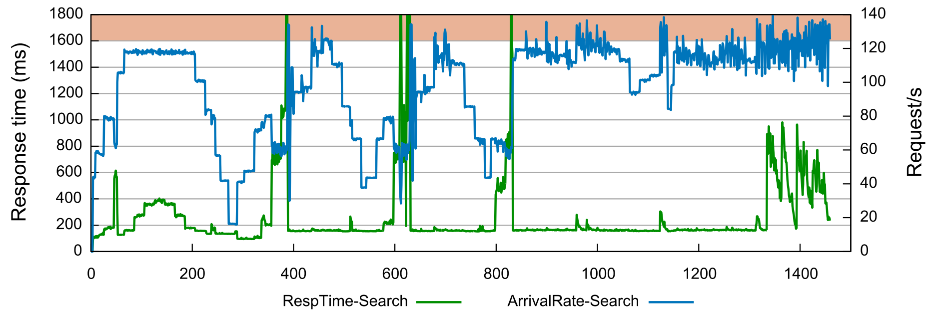
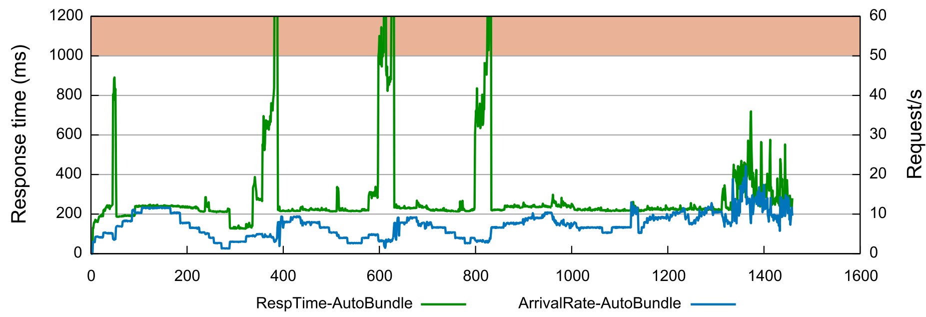


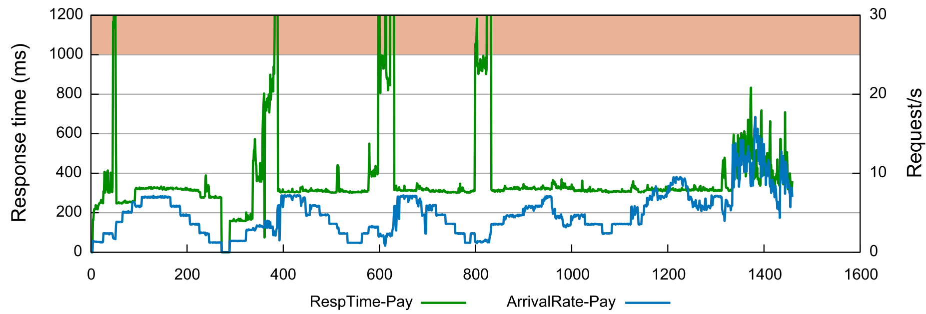
Figure 6.10: ***Direct-ad:*** *Bandwidth rates*





(a) *Response time and arrival rates of Browse and Search*





(b) *Response time and arrival rates of Auto-bundle and Pay*

Figure 6.11: ***Direct-ad***

We can observe in Figures 6.9 and 6.10 that AMS applies a combination of bandwidth and auto-scaling adaptations to cope with the SLA violations such that the profit is maximized according to the optimization module output.

6.4.3.2 Optimized adaptation using indirect advertisement (Indirect-ad) revenue model

Figures 6.12, 6.13, 6.14a, and 6.14b show the results for the optimized management approach using indirect-ad model. As can be seen in the figures, AMS uses a combination of both adaptation types to maximize the profit each time an action is required to be taken when dealing with SLA violations.

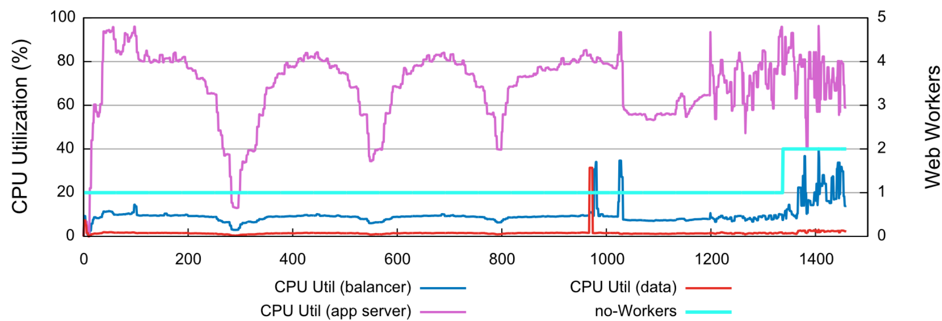


Figure 6.12: ***Indirect-ad:*** *CPU Utilization and no. of workers*

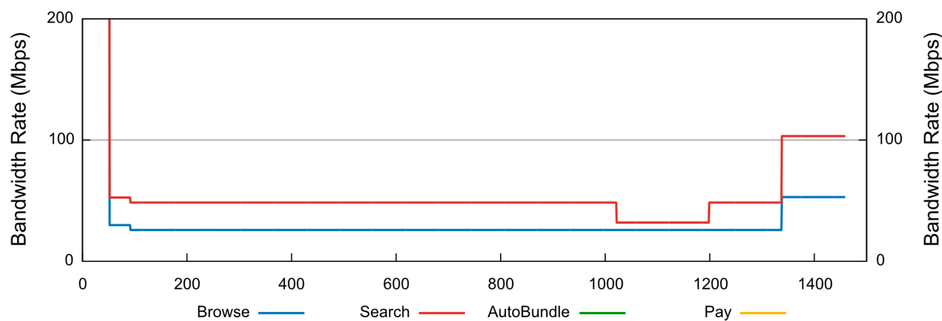
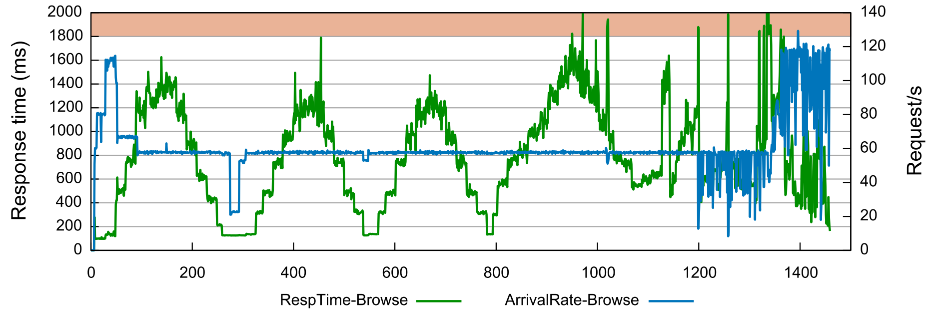
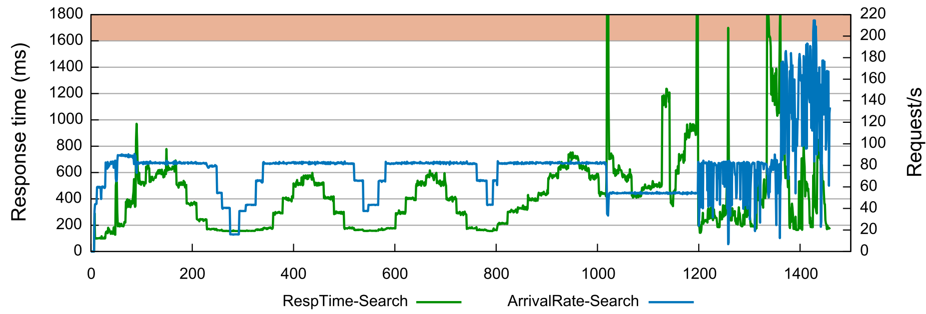
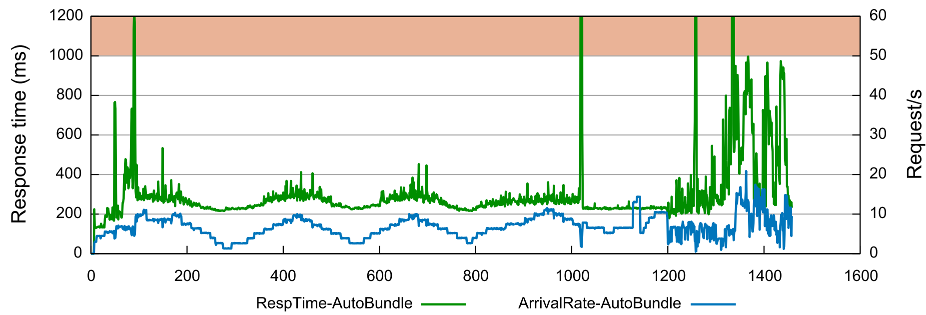


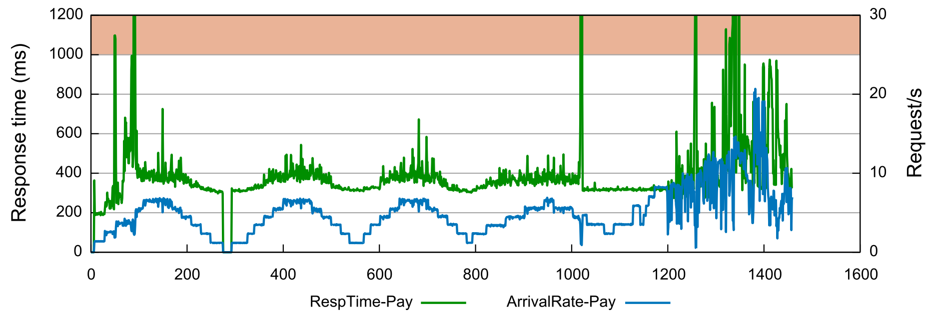
Figure 6.13: ***Indirect-ad:*** *Bandwidth rates*





(a) *Response time and arrival rates of Browse and Search*





(b) *Response time and arrival rates of Auto-bundle and Pay*

Figure 6.14: ***Indirect-ad***

6.4.3.3 Optimized adaptation using no advertisement (No-ad) revenue model

Figures 6.15, 6.16, 6.17a, and 6.17b show the results for the optimized management approach using No-ad model. Using the optimization module online, AMS performs adaptations that maximizes the profit.

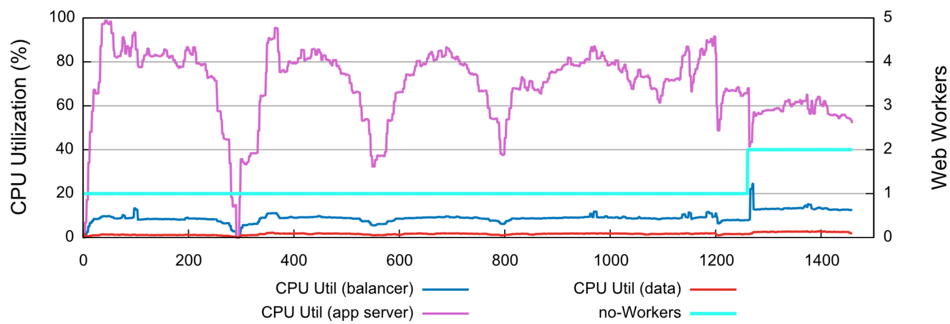


Figure 6.15: ***No-ad:*** *CPU Utilization and no. of workers*

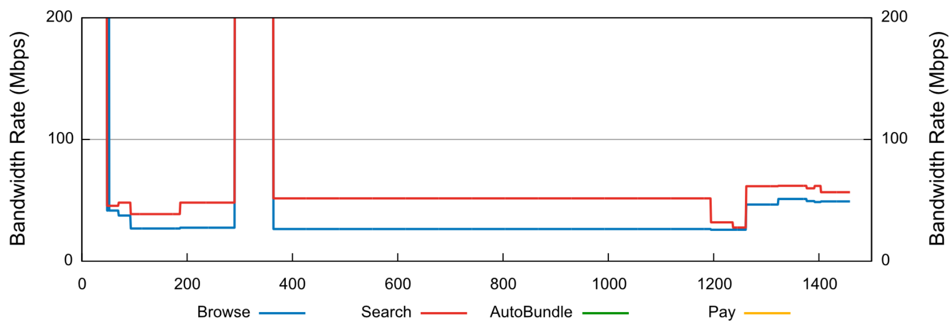
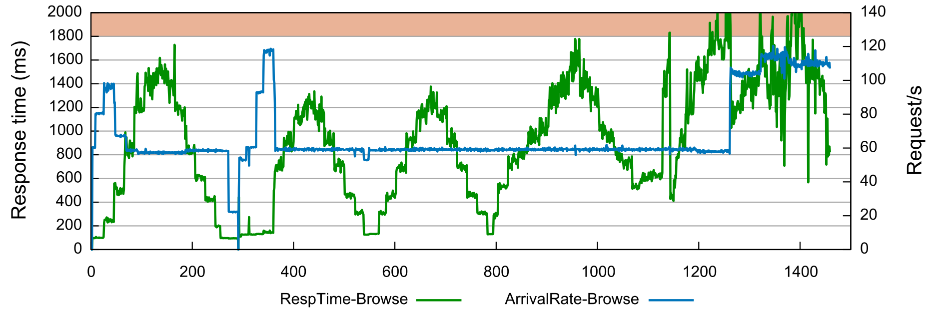
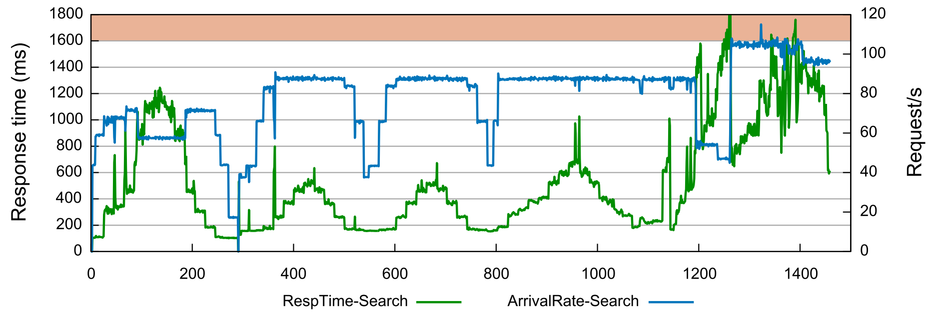
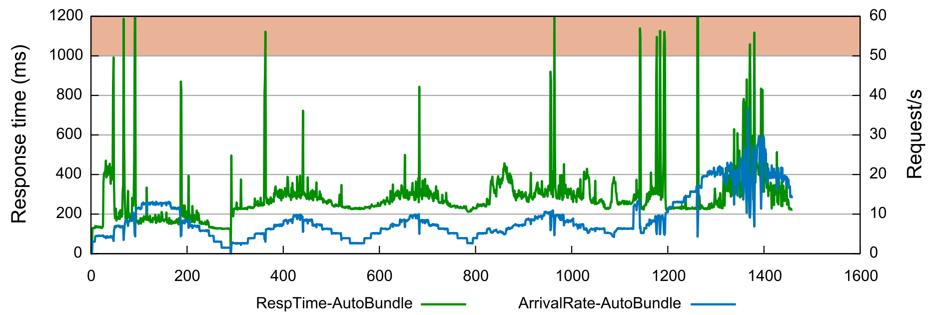


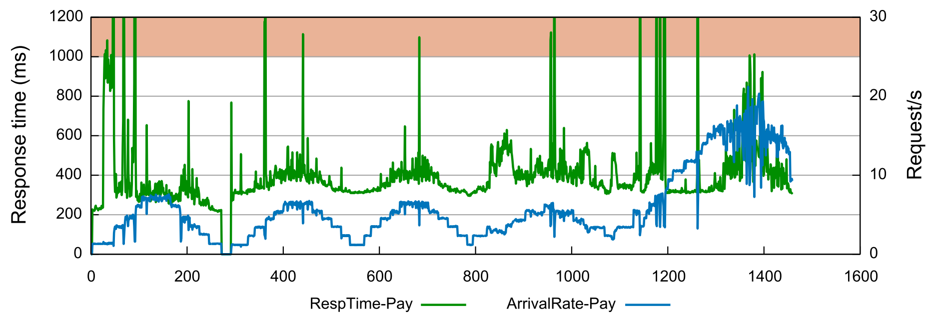
Figure 6.16: ***No-ad:*** *Bandwidth rates*





(a) *Response time and arrival rates of Browse and Search*





(b) *Response time and arrival rates of Auto-bundle and Pay*

Figure 6.17: ***No-ad***

Table 6.6: *Comparison of auto-scaling and bandwidth adaptations in terms of time to effect.*

Metric	VM Auto Scaling	Bandwidth Adjustment
Average time to effect	2.62 min	<< 20s

6.5 Comparison of Approaches

As previously mentioned, one of the advantages of combining network adaptation with auto-scaling is the fast response time of applying network changes. Table 6.6 compares the auto-scaling and bandwidth adaptations in terms of *average time to effect* on response times that were obtained from the experiments. We can see that the result of bandwidth adaptation is seen much faster on the application than auto-scaling approach.[2] That is one of the advantages of using network adaptation in conjunction with auto-scaling for faster reaction time.

We perform a number of analysis on the results of all the experiments. Figures 6.18, 6.19, and 6.20 summarize the statistics that we collected from the results of the experiments. We compare the experiment results with respect to different metrics that are included in the profit optimization problem, namely, revenue (R), data transfer cost (D), instance cost (I), penalty (P) and profit (π).

Figure 6.18 shows the result of the analysis comparing auto-scaling-only, model-based bandwidth approach (BW), and the optimized method using the direct-ad revenue model. Figure 6.19 shows the result of the analysis comparing auto-scaling-only, BW, and the optimized method using the indirect-ad revenue model. Figure 6.20 demonstrates the result of analysis comparing auto-scaling-only, BW, and the optimized method using the no-ad revenue model.

[2] Our measurement precision for monitoring the time-to-effect of bandwidth adaptation is limited to our monitoring interval time which is 20s.

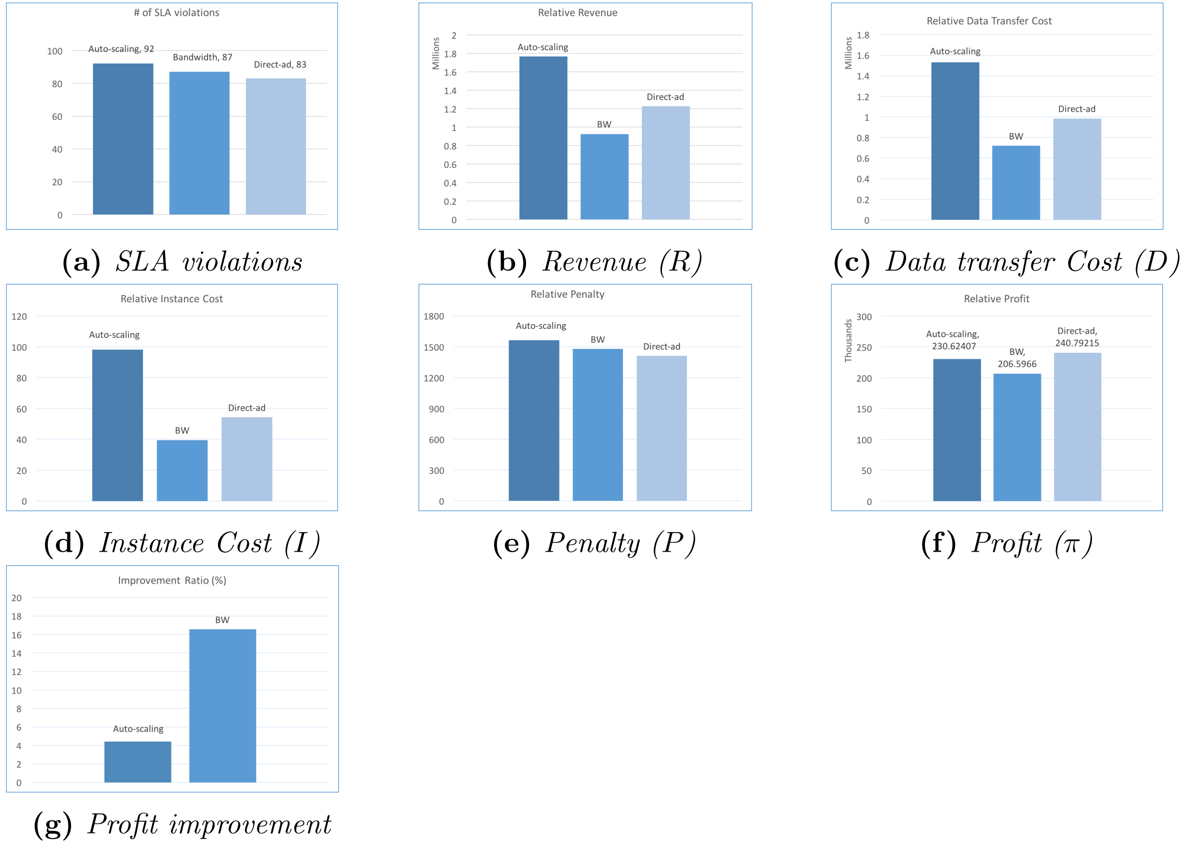
Figure 6.18a shows the number of SLA violations for the three approaches where the optimized method has the lowest number of SLA violations and hence lower penalty in Figure 6.18e. The relative data transfer cost shown in Figure 6.18c is higher in auto-scaling approach because there is no limitation on the bandwidth rates assigned to the scenarios. Also the relative instance cost in Figure 6.18d is higher in auto-scaling approach because this method only scales out the application to maintain the required SLAs. The optimized approach has higher instance cost and data transfer cost compared to BW since it adapts the application using both scaling out and bandwidth adjustments. As can be seen in Figure 6.18f, the optimized approach achieves the highest profit compared to the other two methods where it increases the profit by 4.4% and 16% compared to auto-scaling and BW respectively as shown in Figure 6.18g. Figure 6.19 compares the results from auto-scaling, BW and indirect-ad optimization experiments. As we can observe the profit is maximized in the optimized approach where it is raised by 13% and 6% compared to auto-scaling and BW respectively as depicted in Figure 6.19g.

Figure 6.20 demonstrates that the no-ad optimization approach maximizes the profit compared to auto-scaling and BW by 14.55 times and 0.1 % respectively. The reason for this huge increase in profit compared to the auto-scaling approach is requests in scenarios Browse and Search do not generate any revenue according to no-ad model as shown in Table 6.5; instance hour and bandwidth of the application is consumed without having any corresponding revenue when application is scaled out. This resembles the situations where the application is scaled regardless of the benefits associated with the type of requests. However, the optimized approach uses bandwidth adaptation and slows down the request arrival rates for the scenarios who generates minimum or no

revenue by reducing their allocated bandwidth rates. Therefore, it saves both instance and more importantly data transfer cost as shown in Figure 6.20c and 6.20d. The profit is maximized in the optimized approach as shown in Figure 6.20g. In addition, BW performs close to no-ad optimized approach since in no-ad mode Browse and Search Scenarios do not produce any revenue and their bandwidth rates are decreased similar to BW approach. As a result the costs of instance and data transfer are lowered accordingly.

We can also observe that the percentage of profit improvement is higher in no-ad optimization approach compared to direct-ad and indirect-ad. The reason is that in no-ad model, Scenarios Browse and Search do not produce revenues and in case of SLA violations, they are slowed down, which reduces their request arrival rates due to receiving slow responses. This leads to lower instance and data transfer cost. The level of profit improvement of direct-ad compared to auto-scaling and BW is lower than the other two optimized experiments since all the scenarios generate some amounts of revenue.

Consequently, we can see from the results that combining auto-scaling with network adaptation using NFV and SDN adds more flexibility to applications on the cloud to increase the profit; network adaptations yield faster reaction time which increases application agility to adapt in a more timely and efficient manner. Moreover, dynamic bandwidth adjustment reduces the operational cost of application providers through slowing down the traffic that do not produce considerable revenue. Instead of using simple, rule-based auto-scaling policies, our autonomic management approach dynamically adapts the application both at compute and network levels using models at run-time such that the profit is maximized.



(a) *SLA violations* **(b)** *Revenue (R)* **(c)** *Data transfer Cost (D)*

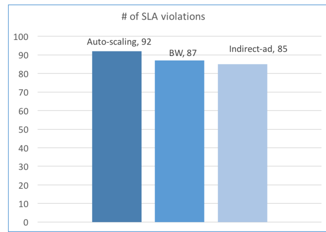
(d) *Instance Cost (I)* **(e)** *Penalty (P)* **(f)** *Profit (π)*

(g) *Profit improvement*

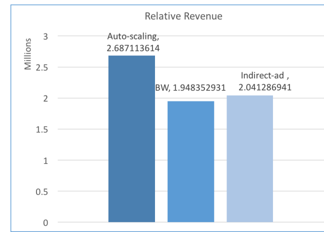
Figure 6.18: *Comparison of auto-scaling, model-based bandwidth approach (BW), and direct-ad optimization .*

6.6 Summary

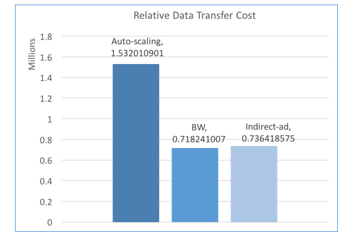
In this chapter, we proposed a novel management mechanism to optimize the profit of cloud web applications using capabilities of SDI. We first developed an optimization problem that takes into account the revenue and price models as well as performance objectives to maximize the profit. Solving the optimization problem online, the management mechanism autonomously adapts the applications at compute and network levels. We implemented and confirmed the applicability of the proposed solution in a hybrid cloud environment of SAVI and AWS. Our results indicated that the proposed



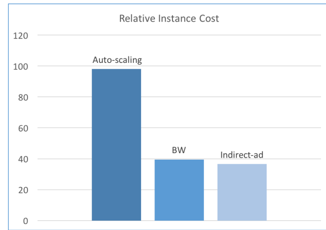
(a) *SLA violations*



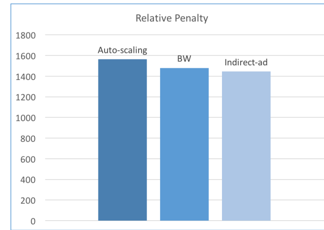
(b) *Revenue (R)*



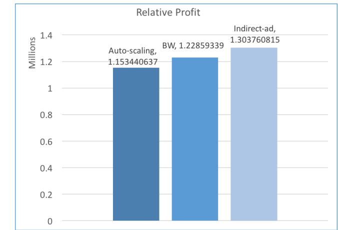
(c) *Data transfer Cost (D)*



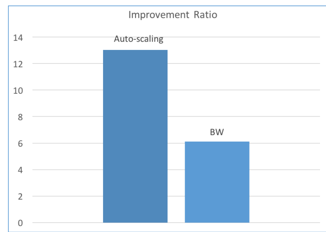
(d) *Instance Cost (I)*



(e) *Penalty (P)*



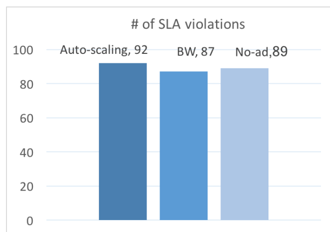
(f) *Profit (π)*



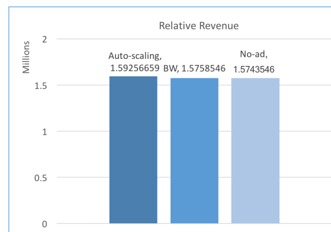
(g) *Profit improvement*

Figure 6.19: *Comparison of auto-scaling, model-based bandwidth approach (BW) and indirect-ad optimization.*

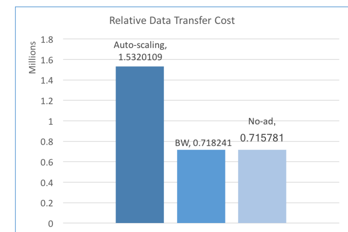
solution is able to improve the profit of a given web application considerably compared to the baseline approaches. Cloud application owners can apply this approach to increase their profit while protecting their auto-scalability feature against exploitations that can hugely harm their business.



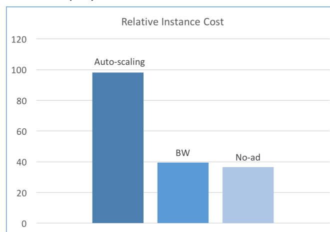
(a) *SLA violations*



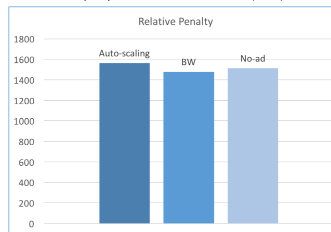
(b) *Revenue (R)*



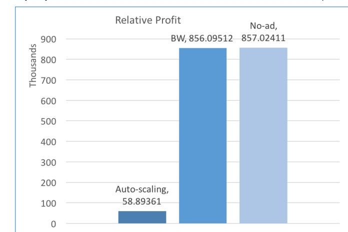
(c) *Data transfer Cost (D)*



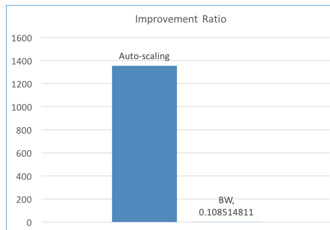
(d) *Instance Cost (I)*



(e) *Penalty (P)*



(f) *Profit (π)*



(g) *Profit improvement*

Figure 6.20: *Comparison of auto-scaling, model-based bandwidth approach (BW), and no-ad optimization.*

Chapter 7

Self-adaptive Applications on Software Defined Infrastructure

In previous chapters, we designed and developed autonomic management systems that can leverage both run-time compute and network programability to meet application non-functional requirements such as performance, cost, and security. In this chapter, we study the efficiency and scalability of SDIs in allowing applications to communicate such network and compute adaptations at run-time. This work is an early assessment of the performance and scalability of a cloud data center SDI in enabling applications to dictate both their computing and networking requirements. To this end, we model an SDI in which applications adapt to their operation conditions by sending requests for dynamic configurations to the cloud. The model can be leveraged by cloud service providers to perform what-if analysis and capacity planning in a systematic manner when they provide enhanced application-awareness services. Each major component of the SDI is modeled via one or more queuing systems so that the complexity of the adaptation process is tackled by a network of queues. Using the model, we obtain the response

time for processing applications' run-time requests under different configurations and parameter settings. We conduct extensive testbed experiments on SAVI cloud [36] to verify the accuracy and fidelity of the model. We show the scalability and bottleneck of the SDI in meeting applications' run-time demands. The results reveal deep insights on efficiency and scalability of the SDI when providing application-aware capabilities.

Hence, our major contributions in this chapter are summarized as below:

- We propose a comprehensive analytical performance model to study the efficiency and scalability of SDIs in fulfilling this new paradigm. The performance model captures sufficient details while maintaining tractability and extendability.

- We carry out extensive testbed experiments on SAVI cloud to verify the analytical model. The results of our modeling quantify the response time for processing application run-time requests within the SDI. Our results reveal the scalability bottleneck of the SDI on SAVI cloud in meeting applications' adaptation requests. The proposed performance model can be leveraged by cloud service providers to perform capacity and bottleneck analysis in a systematic manner when providing full-featured application-aware capabilities.

The remainder of this chapter is organized as follows: Section Section 7.1 presents a conceptual as well as an analytical model of the application adaptation scenarios on the SDI. We explain the validation process of the proposed model in Section 7.2. The results of validation are presented in Section 7.3. Section 7.4 concludes the chapter and states future work.

7.1 Model

In this section we show that a credible model can be used to represent adaptation scenarios on an SDI that we discussed in previous chapters.

We categorize the adaptation cases that applications perform at run-time into two main adaptation classes; the first one involves both compute and network adaptations. In such a scenario, not only compute resources have to be configured on demand, but also the corresponding network flow rules have to be installed on the fly so that the application requirements are properly met. We use application *transposition* to reflect this type of adaptation. Transposition adaptation happens when an application cluster is duplicated elsewhere and the traffic has to be managed between the original application cluster and the duplicated one. An example of such adaption were discussed in Chapter 3 where a shark tank is created in isolation to redirect suspicious traffic on the fly. Duplicating an application cluster and redirection between the two versions can also be performed for other various reasons: cost is another important reason for moving applications due to one cloud data center offering cheaper services than another. Emergency is another factor; flooding is forecasted to impact some part of the country, causing widespread power outages. In such cases, application owners move their running application to other data centers without downtime.

The second main class is the adaptation that only includes networking configurations at run-time (*networking-only* adaptation). In this case, the compute resources are already in place and the application is only interested to do run-time networking configurations including routing instructions and bandwidth adjustments. In Chapters 4, 5 and 6, we proposed to adapt the bandwidth rates assigned to application scenarios to meet performance and cost goals. There exists a third case where applications are only

interested in compute adaptations. As this direction of research has been sufficiently studied in literature [101, 102, 103, 104, 105], we do not focus on this scenario; we are rather interested to study the adaptations that involve networking-level run-time configurations using the emerging SDN technology.

A conceptual model of application adaptation scenarios discussed previously is presented in Fig. 7.1. When the application autonomic manager decides to do adaptation, it uses cloud APIs to communicate its requirements dynamically to the cloud controller. In case of transposition adaptation, the computing resources are firstly provisioned and then the required networking configurations will be dictated to the SDN controller. In case of deprovisioning of resources, networking rules are updated first and then the computing resources will be released. In networking-only adaptation, the cloud controller only sends the required adaptations to the SDN controller. Based on the adaptation scenarios discussed so far, we can identify major operations and components involving in the adaptation scenarios as follows:

- **Cloud controller**: adaptation requests to the cloud SDI first arrive at the cloud controller so that corresponding actions will be determined to realize the adaptations.
- **Application cloning**: This operation includes VM instantiations and application cluster setup, the outcome of which is a running application cluster.
- **State migration**: this operation is carried out when there is a need to copy the state of a VM to another VM.
- **Application de-cloning**: it refers to the process of releasing resources of application clusters.

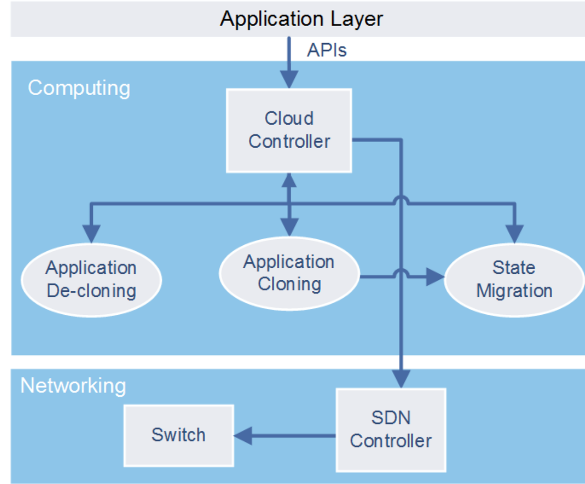


Figure 7.1: *Conceptual model of transposition and networking-only adaptations in SDI*

- **SDN controller**: a machine that programs the networking devices based on commands and policies.

- **Switch**: the forwarding device that forwards the traffic based on flow rules specified by the SDN controller.

Now we present the model of SDI considering two main classes of adaptations including *transposition* and *networking-only*.

We have modeled our SDI using a network of $M/M/1$ queues, single-server queues with Poisson arrivals (i.e, λ_1 to λ_6) and exponentially distributed service rates (i.e., μ_1 to μ_6). In our model, we have two types of external arrivals. The first one is adaptation requests from applications (λ); since the number of potential applications that require adaptations is comparatively small compared to all applications, the adaptation arrival process can be modeled as a Poisson process [1]. The second external arrival is the application flows to the switch (λ_f) which we also assume to be a Poisson process based on [44, 45, 47].

Adaptation requests are sent by the application autonomic manager to the cloud controller. Fig. 7.2 presents the model of application adaptations on the SDI which corresponds to the conceptual model presented in Fig. 7.1. Requests for configuration of compute and network resources are sent to the cloud controller by application autonomic manager. As can be seen in Fig. 7.2, requests are routed based on probabilities $\{P_i,\ 1 <= i <= 7\}$ that characterize the adaptation scenarios. A set of P_is represents a certain adaptation scenario discussed in Section 7.2. Table 7.1 presents the probabilities and their physical meaning.

Probability	**Definition**
P_1	probability by which the request is for compute adaptation
P_2	probability by which the request is for de-clonning the application
P_3	probability by which the request is for application cloning/state Migration
P_4	probability by which the request is for state migration
P_5	probability by which the request is for application cloning
P_6	probability by which the request is for networking-only adaptation
P_7	probability by which the request has already been processed at the compute level

Table 7.1: *Probabilities representing various adaptation requests*

The proposed performance model has three key characteristics. First, it captures sufficient detail in order to maintain the fidelity. Second, it maintains tractability and extendability using a network of queues, each of which realizes different part of the system. Third, it is generic enough to model various adaptation scenarios. Hence, we model the adaptation scenarios with a combination of operations and physical components.

In Fig. 7.2, the queues with dash boxes represent operations while queues with solid boxes are the actual components. Following, we first explain how the model realizes transposition adaptation and then we explain the networking-only adaptation.

Transposition Adaptation: In the transposition scenario, application cloning happens when a new application cluster has to be cloned on a new zone. This process comprises VM instantiations, application setup, and establishment of the logical connections between application cluster components. This process has been modeled by Q_3 in Fig. 7.2. Examples of application cloning include the shark tank creation in Chapter 3. When cloning an application cluster on a new zone, in case of on-going sessions, the run-time state of the application server is also required to be transferred to the destination zone (i.e., state migration modeled by Q_4). An example of this operation happens when the suspicious client is interacting with the original server and it is to be redirected to the shark tank. In this case, when the shark tank is being cloned on the destination zone, the state of the original application server has to transferred to the shark tank application server. Another example is when the application is moved to a new data center while the application is continuously interacting with the clients. Therefore, the state of the application has to be transferred as well. Application de-cloning (modeled by Q_2) takes place when the application cluster has to be removed; a cluster is being moved to a destination zone and required to be de-cloned on the source zone or when the shark tank is not needed anymore, the application cluster in the shark tank has to be de-cloned and resources have to be released.

So far, we have explained the compute run-time adaptations (i.e., the upper part of Fig. 7.2). After the compute resources are in place, the corresponding network flow rules have to be installed on the switch. Note that when de-clonning application clusters, the

reverse order of operations has to be taken (i.e, updating the flow rules first and then releasing the compute resources). The switch (modeled by Q_5) acts as a coordinator to route application traffic based on the application input. After the compute resource are in place, the result of the compute adaptation will be sent to the cloud controller. The cloud controller sends the requests to the SDN controller. When the SDN controller (modeled by Q_6) receives the adaptation requests, it installs flow rules on the switch accordingly. To put this into a perspective, the requests that are sent to the SDN controller may contain the IP addressees of the suspicious clients, original application server, and the shark tank application server so that incoming suspicious traffic is routed from original cluster to the shark tank and vice versa.

Networking-only Adaptation: Our model also reflects the second adaptation scenario where only networking configurations are needed at run-time. This is to cover the scenarios where the required compute resources exist and only networking rules have to be installed. The networking-only scenario is captured when $P_1 = 0, P_6 = 1$ (see Fig. 7.2). Therefore, the adaptation requests only go through the cloud controller, SDN controller, and the switch.

We have two types of requests arriving to the cloud controller (Q_1). One is transposition adaptation requests that are first processed by provisioning compute resources; then the results are sent back to the cloud controller so that they are forwarded to SDN controller for network processing. The second type of requests are those that only need adaptation at networking level as the required compute resources already exist. We assume that adaptation requests are generated with mean arrival rate of λ. Adaptation requests will be sent to Q_2, Q_3 and Q_4 for application-cloning and de-cloning with probabilities P_1–P_5. Then the cloud controller sends the required adaptations to the

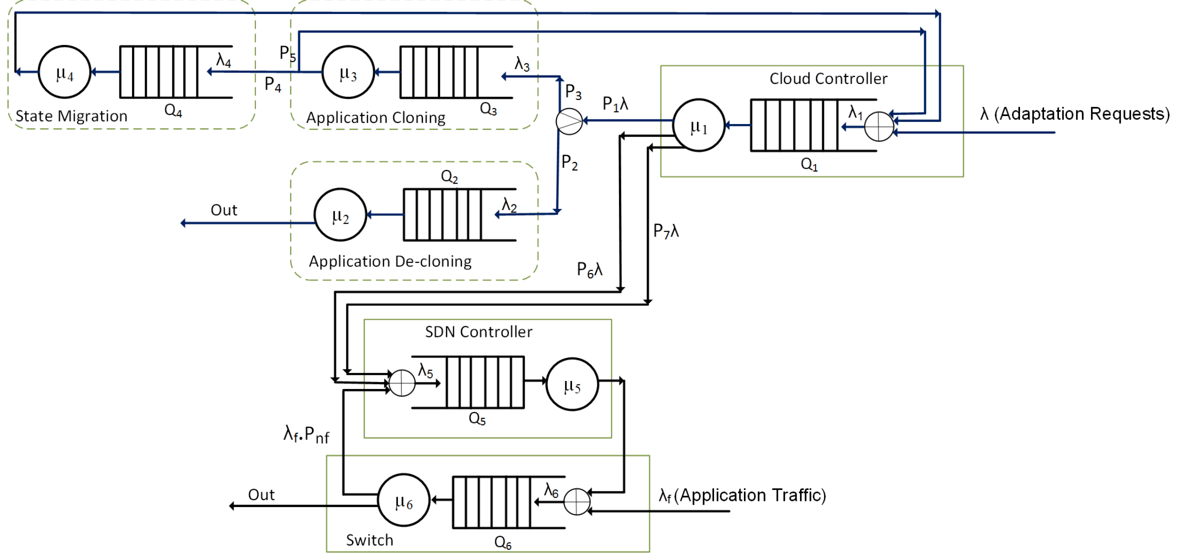


Figure 7.2: *Model of adaptation scenarios on an SDI*

SDN controller so that flow rules are installed on the switch (proactive flow installation). In addition, SDN controller is also responsible for installing rules for a fraction of flow arrivals to the switch with mean rate of λ_f for which the switch does not have any forwarding rule (reactive flow installation). In other words, two types of jobs are to be processed at the SDN controller: one is to install flow rules for adaptation requests, and another one is to install forwarding rules for the incoming application traffic with no flow rule entry on the flow table of the switch.

We can obtain the response time of adaptation requests based on Little's law. For response time of adaptation requests (AR), we have:

$$\begin{aligned} RT_{AR} = & \frac{1}{\mu_1 - \lambda_1} + \frac{P_1 * P_2}{\mu_2 - \lambda_2} + \frac{P_1 * P_3}{\mu_3 - \lambda_3} \\ & + \frac{P_1 * P_3 * P_4}{\mu_4 - \lambda_4} + \frac{P_1 * P_3}{\mu_1 - \lambda_1} + \frac{P_7 + P_6}{\mu_5 - \lambda_5} \end{aligned} \tag{7.1}$$

Where

$$\lambda_1 = \lambda + (P_1 * P_3 * \lambda) \tag{7.2}$$

$$\lambda_2 = P_1 * P_2 * \lambda \tag{7.3}$$

$$\lambda_3 = P_1 * P_3 * \lambda \tag{7.4}$$

$$\lambda_4 = P_1 * P_3 * P_4 * \lambda \tag{7.5}$$

$$\lambda_5 = ((P_6 + P_7) * \lambda) + (\lambda_f * P_{nf}) \tag{7.6}$$

$$\lambda_6 = (\lambda_f * (P_{nf} + 1)) + \lambda * (P_6 + P_7) \tag{7.7}$$

where μ_i and λ_i represent the service rate and arrival rate of Q_i respectively. P_is show the routing probabilities between queues depending on the specification of the adaptation requests which can be calculated from 7.8.

$$P_1 = 1 - P_6, \quad P_2 = P_1 - P_3, \quad P_4 = 1 - P_5, \quad P_7 = P_1 \tag{7.8}$$

In Equations 7.6 and 7.7, P_{nf} represents the probability of having no flow rules for the application flow arrivals to the switch with rate of λ_f. In calculating the packet response time, we considered OpenFlow for the communication standard between the switch and SDN controller. Therefore, with probability $1 - P_{nf}$, there are forwarding rules for the flow arrivals (λ_f) and the packets only travel through the switch [47]. With probability P_{nf}, there is no entry for the flow arrivals, therefore, the first packet of each flow will travel through the switch, then to the controller and back to the switch. Hence,

the mean response time of packets can be calculated as

$$RT_{pkt} = \frac{1 + P_{nf}}{\mu_6 - \lambda_6} + \frac{P_{nf}}{\mu_5 - \lambda_5} \tag{7.9}$$

7.2 Implementation

In this section, we describe the implementation and evaluation of our analytical modeling and test bed experiments. We have implemented the analytical performance model in Python using simpy, scipy, and numpy libraries[1]. Table 7.2 presents the parameters that are inputs to our model. Inputs to the model include application traffic arrival rate (λ_f), adaptation request arrival rate (λ), service rates of the queues (μ_is), and probabilities (P_is). Outputs are the response time for processing adaptation requests and packets by which scalability and bottleneck of the SDI are studied. λ_f and λ depend on the applications being considered. To measure the service times of queues in the model, we performed each operation (represented by a qeueu in the model) multiple times on SAVI cloud and recorded the average time it takes to do each operation. For instance, we cloned and de-cloned a three-tier application cluster multiple times and measured the average time for performing each operation. For SDN controller and switch service times, we measured the time for processing packets and installing flow rules multiple times and averaged the values. The measured parameters are given to the model as inputs to solve the model. Probabilities P_is depend on the adaptation scenario that applications perform.

To verify the validity of the analytical model, we have developed the two major adaptation scenarios including networking-only and transposition scenarios on SAVI.

[1] www.scipy.org

Table 7.2: *Exogenous parameters of the model measured on SAVI cloud*

Parameter	Value	Unit
λ_f	400-12,00	(flows/sec)/application
λ	1-400	(req/hour)/application
μ_1	94	req/sec
μ_2	0.01	req/sec
μ_3	0.02	req/sec
μ_4	0.2	req/sec
μ_5	416	req/sec
μ_6	205493	req/sec
P_i	[0,1]	NA

In our experiments, we imitate different distributions of adaptation requests (i.e., various values for P_is) that are sent to the cloud controller; we aim to cover network-intensive, computing-intensive and a combination of both adaptation scenarios that reflect various scenarios. Table 7.3 represents the scenarios and the corresponding probability distributions. Note that other probabilities (i.e., P_3, P_5, P_6, P_7) are dependent probabilities that can be calculated from P_1, P_2, P_4 (see Equation 7.8).

In Table 7.3, with regards to adaptation examples discussed thus far in the thesis, Scenario A represent the case of shark tank where the states of VMs are also transferred to the destination cloud. Scenario B may represent situations where shark tank is created less frequently and there is no suspicious users interacting with the application at the time of shark tank instantiation, therefore no VM state migration is needed, hence $P_4 = 0$. Scenario C is similar to Scenario A but the application needs compute adaptation less frequently. Scenario D can be an example of shark tank creation where the application creates and destroys the shark tank more frequently than scenario B and state migration happens as suspicious users are continuously interacting with

the application, hence $P_4 = 0.2$. Scenario E represents scenarios where applications are only interested to change the networking. This represents the dynamic bandwidth adjustments. Also Scenario E can be an example of redirecting suspicious clients to an existing shark tank where no compute adaptation is needed, hence $P_1 = 0$.

Scenario	P_1	P_2	P_4	**Adaptation Type**
A	0.00028	0.5	0.8	Transposition
B	0.0002	0.5	0	Transposition
C	0.00014	0.5	0.5	Transposition
D	0.00042	0.5	0.2	Transposition
E	0	0	0	Networking-only

Table 7.3: *Experiment scenarios*

To implement the adaptation scenarios presented in Table 7.3, we set up the experiments with two networks where resources have to be dynamically provisioned/deprovisioned between the source and destination zones and flow rules have to be installed on the forwarding devices so that the traffic is routed between the two zones depending on the application requirements. Fig. 7.3 presents our experiment setup on SAVI cloud.

We employ Virtual Extensible LAN (VXLAN) overlay networks to implement the solution. A VXLAN can create arbitrary Layer 2 (L2) or Layer 3 (L3) networks by encapsulating L2 frames in L3 UDP packets. We setup our experiment by deploying three types of overlay networks that host (1) original application clusters, (2) destination clusters, and (3) clients. We use the same application that is used in previous chapters.

In order to increase the load on the SDN controller in processing and installing reactive flows, we set a large number of clients to interact with applications. To this end,

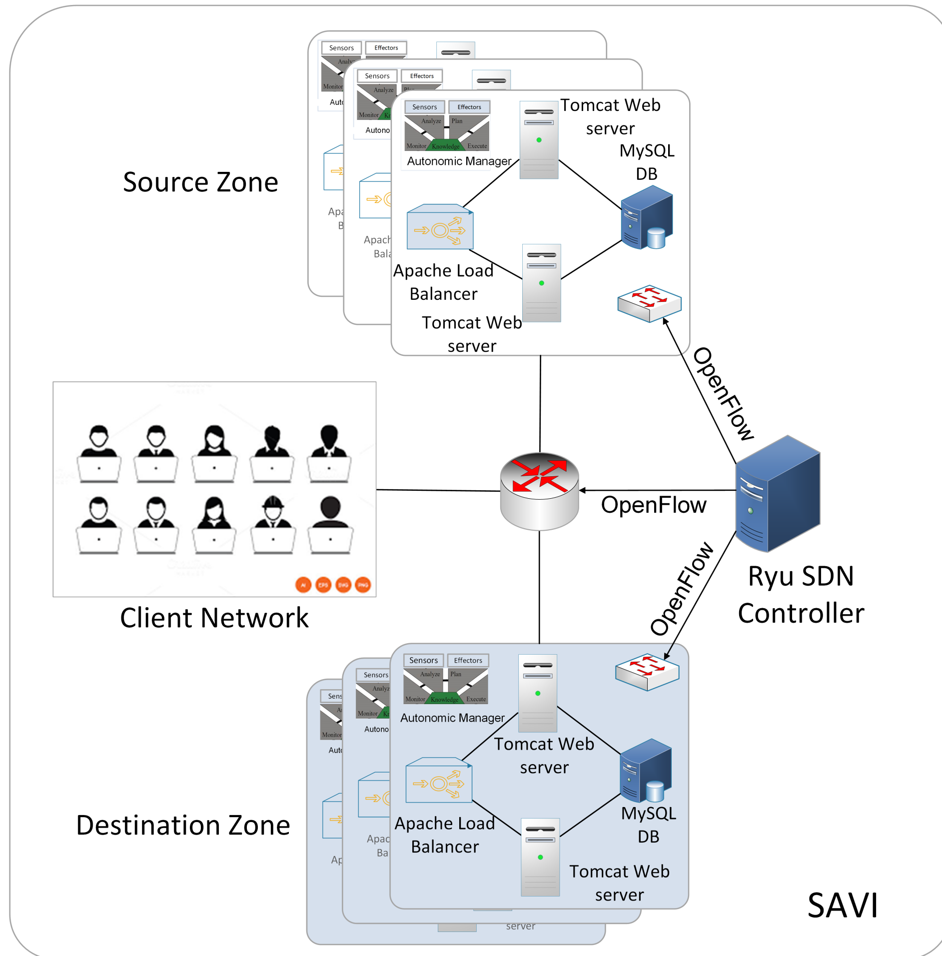


Figure 7.3: *Experiment setup on SAVI cloud*

we set up a client overlay network on SAVI that is fully separated from the application cluster networks. We used 8 VMs on the client network and on each VM, we setup 40 VXLAN links. Each link is connected to a port with an IPv4 address representing a client (i.e., 320 clients in total). We then add routing entries to the Linux kernel routing table for each interface so that each interface (i.e., client) sends/receives traffic independently from other clients on the same VM to avoid ARP flux problem [80]. This way, the SDN controller is responsible for setting up flow rules for each client interacting with the application in addition to processing application adaptation requests. We developed

Table 7.4: *Specification of VMs on SAVI cloud*

Flavor	Virtual CPU (VCPU)	Memory	Disk
Small	1	2GB	20GB
Medium	2	4GB	40GB
Large	4	8GB	80GB

our own workload generator that clients use to send custom http requests to the web applications.

7.3 Experiment

Table 7.4 shows the flavor of the VMs and Table 7.5 represents the node specification in our testbed experiments on SAVI cloud. Ryu SDN framework [76] was installed on a small machine specified in Table 7.4 and Open Virtual Switch (OVS) was used in overlay networks. Our SDN controller uses OpenFlow v1.0.0 to communicate with the switches. We designed and implemented a multi-threaded SDN controller application that can process packets and adaptation in parallel. We used RabbitMQ [78] for the communication between cloud controller, SDN controller and applications' autonomic managers.

Fig. 7.4 depicts the packet response time in our model in Scenario C. We can see from Fig. 7.4, as P_{nf} increases, more packets will be sent to the SDN controller for processing. Therefore, we can see that the response time of the packets at the same number of applications is higher in larger P_{nf}. Furthermore, we can observe that the SDN controller gets saturated at higher P_{nf} with lower number of applications. For example, when $P_{nf} = 0.2$, SDN controller can support 6 applications that perform adaptations.

Table 7.5: *Experiment specification on SAVI cloud*

Networks or nodes	**No. of VMs**	**Flavor**	**Software**
Client network	8 (320 clients)	6 Small & 2 Medium	Custom HTTP Workload Generator
Original Application Clusters	4-7	Large	Apache Load Balancer, Custom Tomcat Web Apps, MySQL
Destination clusters	On Demand	On Demand	Apache Load Balancer, Custom Tomcat Web Apps, MySQL
Virtual Switches	19	Small	OVS
SDN Controller	1	Small	Ryu Custom Controller Application

However, at $P_{nf} = 0.05$ the response time increases slightly and the SDN controller can support the maximum number of applications we set in our model without being saturated. Therefore, results show the scalability issue of the SDN controller where the SDN controller capacity is limited by the number of applications. Cloud service providers can use the model to size their SDN controller with respect to the maximum number of applications performing adaptations. In order to tackle this scalability issue, a more powerful machine should be used to host the controller (i.e., scaling up the SDN controller) as we used a machine with a limited capacity (see Table 7.4 and 4.2.) Alternatively, multiple controllers can be used to scaling out the SDN controller.

Fig. 7.5 shows the amount of time it takes for adaptation requests (ARs) to be processed on the cloud including at the computing level (i.e., $Q_1 - Q_4$) and at networking level (i.e., Q_5, Q_6) when $P_{nf} = 0.01$. Four scenarios (A-D) comprising different values of P_1, P_2, P_4 are shown in Fig. 7.5. As an example, in Scenario A, applications send

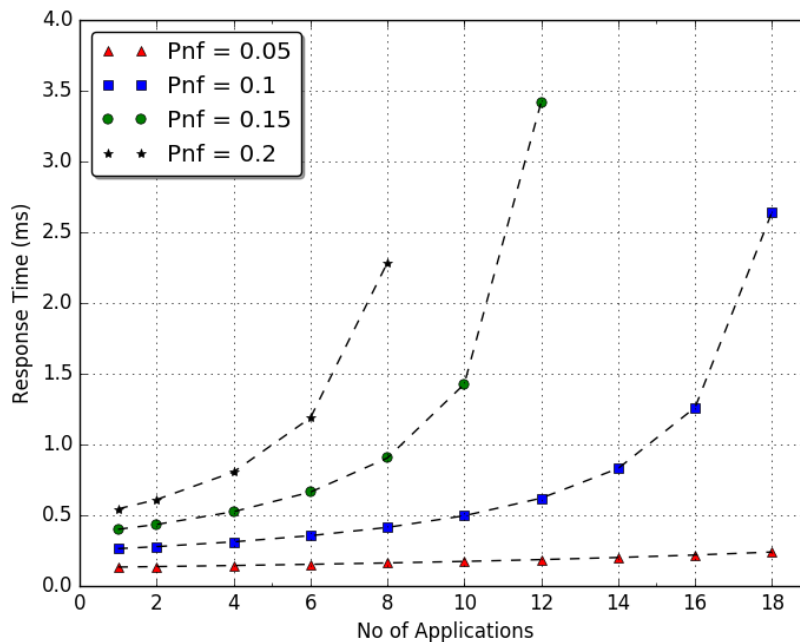


Figure 7.4: *Packet response time*

requests for compute resource adaptation once every hour (i.e, $P_1 = 2.8 * e^{-4}$) half of which are for application cloning where 80% (i.e., $P_4 = 0.8$) of such requests include state migration. Another half of compute adaptation requests are for de-provisioning resources (i.e. $P_2 = 0.5$)

As can be seen in Fig. 7.5, when the number of applications increases, the AR response time raises exponentially in all 4 scenarios. Scenario B has the minimum response time as most of the requests include networking-only adaptations (i.e., network-intensive). Also from the requests that include compute adaptations, no application-cloning along with state migration is happening in Scenario B, which is the most time-consuming job. Scenario C has the lowest response time after scenario B because the number of requests for compute adaptation is less compared to scenario A and D. Scenario D reaches the

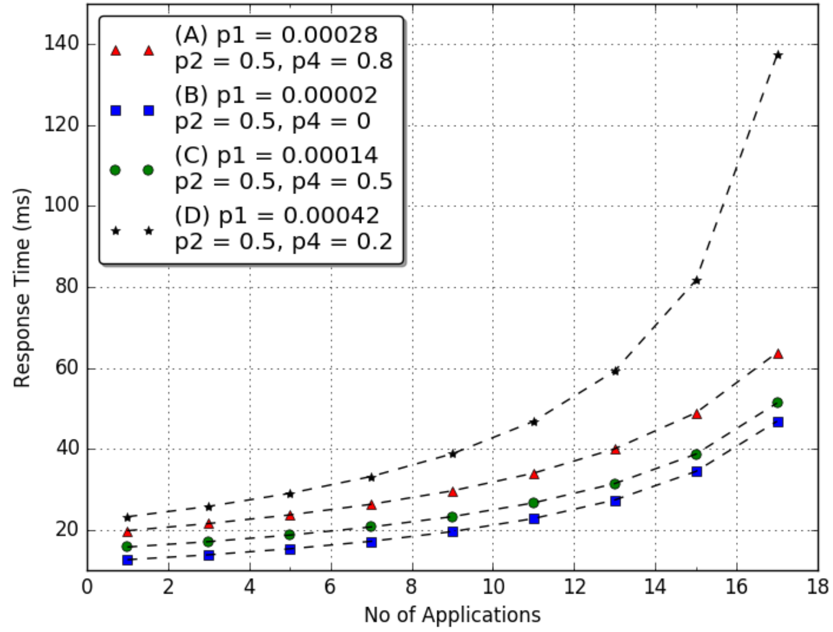


Figure 7.5: *Application adaptation request (AR) response time*

saturation point faster (i.e., in less number of applications) because it has the maximum number of requests that include compute adaptations (compute-intensive). We can see that in Scenario D, up to 15 applications can be supported before the saturation point. Also, it can be noticed that the network-intensive adaptation scenario has the lowest response time while the compute-intensive adaptation scenario yields the highest response time.

Fig. 7.6 illustrates the results from the analytical model and experiment for one application with varying arrival rate for Scenario C where

$$\lambda_f = 1000 \, \text{flows/sec}, \; P_{nf} = 0.01$$

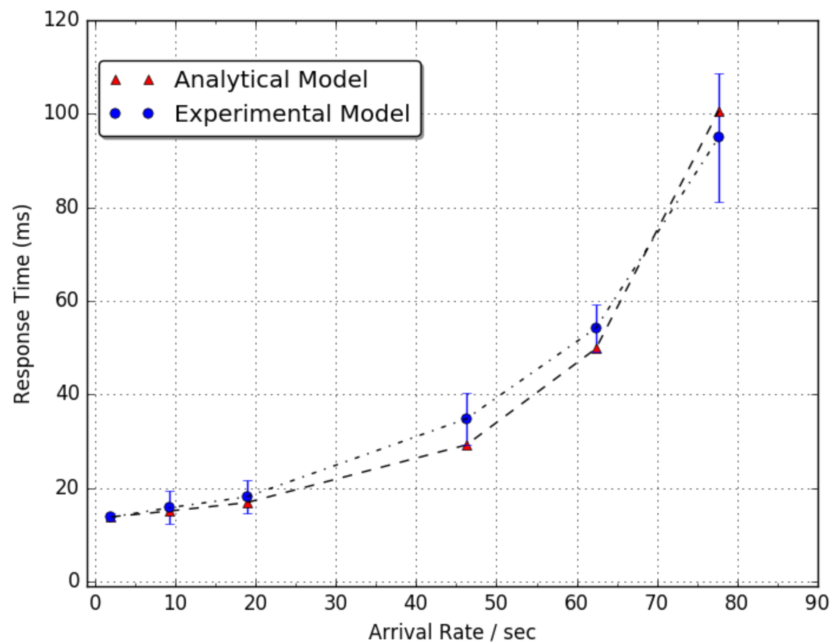


Figure 7.6: *Application adaptation request (AR) response time: analytical model vs. experiment, Scenario C,* $\lambda_f = 1000 flows/sec, P_{nf} = 0.01$

We ran the experiments 6 times and calculated the standard deviation. We can see that the result from the experiment matches that of the analytical model (9% difference in worst case scenario.)

In order to have a finer-grained performance evaluation of our SDN controller, we did an experiment and evaluated the SDN response time for processing adaptation requests to redirect the traffic between the source zone and destination zone as well as bandwidth adjustments (Networking-only adaptation) in presence of incoming traffic flows. Fig. 7.7 shows the result from both experiment and the model. The result is for Scenario E where

$$\lambda_f = 1000 \text{ flows/sec}, \ P_{nf} = 0.05$$

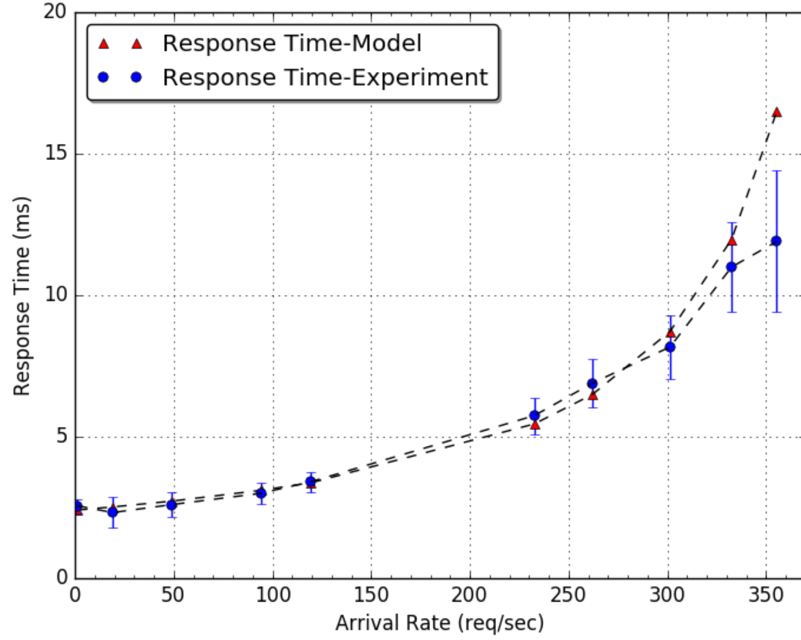


Figure 7.7: *Application adaptation request (AR) response time at the SDN controller: analytical model vs. experiment*

in which only networking rules are to be installed for existing compute resources (networking-only adaptation) as this helps us to have a closer look at the performance of the SDN controller. From Fig. 7.7, we can observe that the response time for 320 req/sec is under 10 milliseconds and for up to 310 req/sec the model and experiment match well. However, with arrival rate higher than 310 req/sec, the analytical model starts to deviate from that of the experiment and it is reaching the saturation point where $\lambda 5 \leq \mu_5 = 416\,\text{req/sec}$. This is due to the fact that when the number of arrivals increases, the system shows higher service rate than the original rate set in the analytical model.

Hence, cloud service providers can use the model to see how many adaptive applications can be supported by the whole infrastructure given system properties. They

can investigate the bottleneck of their infrastructure when providing application-aware capabilities; in Scenario D (compute-intensive) in Fig. 7.5, for example, only up to 8 applications can do adaptations before the response time increases sharply. Also, cloud providers can use the model to predict the response time when regulating SLA with application owners.

7.4 Summary

In this chapter, we modeled and evaluated the performance of an SDI by using two main classes of application adaptations that were proposed in engineering self-adaptive applications. We derived the response time of adaptation requests as well as traffic flows analytically and verified it through experiments on SAVI cloud. We then specifically investigated the efficiency and scalability of the SDN controller. The model can be leveraged by cloud service providers to do what-if and bottleneck analysis in a systematic manner when they provide application-aware capabilities.

Future work includes modeling and evaluating the performance and scalability of an SDI where there are various applications running on dedicated overlay network that is managed by a SDN controller. Studying a hierarchical SDN control model within such an SDI is another direction for future research.

Chapter 8

Conclusion

Cloud computing is a transformative technology that facilitates flexibility, efficiency and competitiveness; users can scale services to fit their needs, customize applications and access cloud services from anywhere with an internet connection. Enterprise users can get applications to market quickly, without worrying about underlying infrastructure costs or maintenance.

With cloud business model, companies are looking for methods to minimize the operational expenses while maintaining performance goals through optimizing the resource consumption and increasing the efficiency in cloud. Due to unlimited available resources, cloud applications become an attractive target for DDoS attacks that can affect user experience and increase the operational cost by causing unnecessary scaling.

Autonomic management systems (AMS) can be designed and developed to continuously monitor applications in cloud, analyze the monitored data, plan and execute corrective actions to maintain the non-functional requirements. Auto-scaling is one of the most common adaptations that an AMS can perform to deal with the fluctuating workload. Auto-scaling may not always be the most economic and efficient course of

actions; from economic perspective, some traffic might not be as beneficial such as traffic with low monetary impacts or illegitimate traffic such as application layer distributed denial of service (DDoS) attacks. Scaling in such circumstances may result in higher cost without any corresponding effect on the revenue. In case of attack traffic, early detection may prevent unnecessary scale out actions. Nonetheless, large false positives associated with many detection mechanisms, lead to customer dissatisfaction and decline in service revenue, a risk that many companies strive to avoid.

Network level mitigation actions are unable to detect and mitigate application layer DDoS attacks because of lack of knowledge about the application characteristics. Examples of such attacks are low and slow application layer DDoS (LSDDOS) attacks that are low in volume but impose large processing overhead to applications. Such attacks go unnoticed by network level detection rules offered by cloud providers that often detect suspicious heavy traffic.

New types of attacks have been introduced with emergence of blockchain where the CPU power of cloud applications are stolen for mining purposes which eventually triggers auto-scaling resulting in huge cost to the application owner.

In terms of efficiency, it may take a non-negligible time until the new node effects due to reconfiguration cost. Also in big data applications, scaling become harder because of data consistency, data replication, and job rescheduling problems.

Therefore, other mechanisms are needed to complement auto-scaling to enhance the way applications operate on the cloud to deal with the mentioned challenges. To this end, in this thesis, we investigated methods and mechanisms to use dynamic network configuration (network adaptation) in addition to auto-scaling (compute adaptation) to design self-managing applications that meet performance, cost, and security objectives

of cloud applications. We take advantage of SDN and NFV techniques along with auto-scaling to engineer autonomic management systems where there is a feedback loop between operation and analysis to derive best course of actions that guarantees the desired outcome.

We proposed CAAMP, an autonomous DDoS mitigation solution where resources from private cloud are used to host suspicious traffic in a shark tank. The shark tank protects the applications against extra cost and bad user experience that are due to DDoS traffic received on public cloud. Also, CAAMP reduces the number of false positives by isolating the suspicious traffic for further analysis; suspicious users continue to receive services until final decision is made. Being inspired by CAAMP, we proposed an autonomic management system with a game-theoric model that takes into account economic weights of the mitigation actions to adapt in a less-expensive manner.

We proposed a model-based AMS that adapts bandwidth rates of application scenarios at the application layer to complement auto-scaling. We first showed how bandwidth adaptation can delay auto-scaling to reduce the cost while still meeting performance requirements through a search-based heuristic model. We then improved the quality of adaptation by replacing the search-based algorithm with a machine learning model to speed up the reaction time. Consequently, we proposed an AMS that maximizes the profit given the workload type, revenue model, price model as well as adaptation options. The optimizer uses a number of scalable online machine-learning models to estimate parameters of a potential adaptation and then choose the most profitable one. The resulting adaptation can be a scaling event, bandwidth adjustment or a combination of both adaptation options and their quantities.

We proposed an analytical model to represent an SDI in presence of dynamic compute

and network adaptations that are performed by adaptive applications. We used queuing networks to model the system and calculated response time of realizing adaptation requests, given the capacity of the SDI. The model can be leveraged by cloud service providers to perform what-if and bottleneck analysis in a systematic manner when they provide application-aware capabilities.

Our proposed solutions are all implemented and evaluated on real cloud environment to ensure their applicability and potential use in real world. Also we provide a plug-n-play framework where various applications can be deployed dynamically on an overlay network managed by their own dedicated SDN controller. This feature makes our solution cloud-agnostic which makes it attractive to application providers that do not want to be locked in to any specific cloud provider. Also due to a modular design, various models can be tested to improve the quality of adaptations.

Future Work

In managing the bandwidth rates of scenarios, our AMS adapts the bandwidth rates only at the proxy node within the application cluster. Investigating other places within the application cluster to adapt the bandwidth is an interesting direction for future research. The autonomic manager may evaluate each possible location online and choose the one that best satisfies the desired objectives.

We assumed that web workers are homogeneous and the load balancer evenly distributes requests to the workers. A possible direction to future research is to develop machine learning models considering heterogeneous workers and other distribution algorithms such as weighted round robin (WRR).

Reactive mechanisms are simple and effective if the user understands their resource

requirements to a great extend. However, the reaction time may affect the performance especially in sudden traffic bursts. Predictive approaches should predict future workloads over a reasonable time to be effective. We used domain knowledge expertise about the application to restrict the space state of workload when training the machine learning models offline. A workload predictor model will make our solution predictive by anticipating the workload ahead of time. Hence, one direction of future research would be to compare the two approaches in terms of cost, performance, and the overall profit.

The performance model of SDI can be extended to include other types of adaptation scenarios as well as topologies. For example, we can have multiple switches to install the network adaptation requests. Also resolving conflicts among different network adaptations can be a topic for further investigation.

Bibliography

[1] H. Khazaei, J. Misic, and V. B. Misic, "Performance analysis of cloud computing centers using m/g/m/m+r queuing systems," *IEEE Transactions on parallel and distributed systems*, vol. 23, no. 5, pp. 936–943, 2012.

[2] W. Vogels., "Day 2 keynote. presentation at re:invent, amazon web services," 2012.

[3] M. Shtern, M. Smit, B. Simmons, and M. Litoiu, "A runtime cloud efficiency software quality metric," in *Companion Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE Companion 2014, 2014, pp. 416–419.

[4] M. Jensen, J. Schwenk, N. Gruschka, and L. Iacono, "On technical security issues in cloud computing," in *Cloud Computing, 2009. CLOUD '09. IEEE International Conference on*, Sept 2009, pp. 109–116.

[5] Q. Yan, F. Yu, Q. Gong, and J. Li, "Software-defined networking (sdn) and distributed denial of service (ddos) attacks in cloud computing environments: A survey, some research issues, and challenges," *Communications Surveys Tutorials, IEEE*, vol. 18, no. 1, pp. 602–622, Firstquarter 2016.

[6] T. Xing, Z. Xiong, D. Huang, and D. Medhi, "SDNIPS: Enabling Software-Defined Networking based intrusion prevention system in clouds," in *10th International Conference on Network and Service Management (CNSM) and Workshop*, Nov 2014, pp. 308–311.

[7] Y. Brun, R. Desmarais, K. Geihs, M. Litoiu, A. Lopes, M. Shaw, and M. Smit, "A design space for self-adaptive systems," in *Software Engineering for Self-Adaptive Systems II*, ser. Lecture Notes in Computer Science, R. de Lemos, H. Giese, H. Mller, and M. Shaw, Eds. Springer Berlin Heidelberg, 2013, vol. 7475, pp. 33–50.

[8] L. Zhao, S. Sakr, and A. Liu, "A framework for consumer-centric sla management of cloud-hosted databases," *IEEE Transactions on Services Computing*, vol. 8, no. 4, pp. 534–549, July 2015.

[9] Amazon Inc., "Aws auto scaling," https://aws.amazon.com/autoscaling/.

[10] J. V. Bibal Benifa and D. Dejey, "Rlpas: Reinforcement learning-based proactive auto-scaler for resource provisioning in cloud environment," *Mobile Networks and Applications*, Jan 2018.

[11] A. Evangelidis, D. Parker, and R. Bahsoon, "Performance modelling and verification of cloud-based auto-scaling policies," in *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, May 2017, pp. 355–364.

[12] J. Jiang, J. Lu, G. Zhang, and G. Long, "Optimal cloud resource auto-scaling for web applications," in *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, May 2013, pp. 58–65.

[13] A. Gandhi, P. Dube, A. Karve, A. Kochut, and L. Zhang, "Adaptive, model-driven autoscaling for cloud applications," in *11th International Conference on Autonomic Computing (ICAC 14)*, Philadelphia, PA, 2014, pp. 57–64.

[14] D. Olenick, "Cryptocurrency mining attacks increasing exponentially, no end in sight," https://www.scmagazine.com/cryptocurrency-mining-attacks-increasing-exponentially-no-end-in-sight/article/750761/, year=2018.

[15] A. Bremler-Barr, E. Brosh, and M. Sides, "Ddos attack on cloud auto-scaling mechanisms," in *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*, May 2017, pp. 1–9.

[16] H. Khazaei, C. Barna, N. Beigi-Mohammadi, and M. Litoiu, "Efficiency analysis of provisioning microservices," in *Cloud Computing Technology and Science (CloudCom), 2016 IEEE International Conference on.* IEEE, 2016, pp. 261–268.

[17] "Juniper networks," https://www.juniper.net/us/en/.

[18] CENGN, "Enabling next generation networks," https://www-01.ibm.com/ibm/cas/canada/.

[19] ——, "Enabling next generation networks," https://www.cengn.ca/about-us/.

[20] "Telus," https://www.telus.com/en/careers/graduate-technology-leadership-program/program-details.

[21] R. Calinescu, Y. Rafiq, K. Johnson, and M. E. Bakir, "Adaptive model learning for continual verification of non-functional properties," in *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering*, ser. ICPE 14. ACM, 2014, pp. 87–98.

[22] M. Salehie and L. Tahvildari, "Self-adaptive software: Landscape and research challenges," *ACM Trans. Auton. Adapt. Syst.*, vol. 4, no. 2, pp. 14:1–14:42, May 2009.

[23] S. Dobson, S. Denazis, A. Fernandez, D. Gatti, E. Gelenbe, F. Massacci, P. Nixon, F. Saffre, N. Schmidt, and F. Zambonelli, "A survey of autonomic communications," *ACM Trans. Auton. Adapt. Syst.*, vol. 1, no. 2, pp. 223–259, Dec. 2006.

[24] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, Jan 2003.

[25] C. Li, B. Brech, S. Crowder, D. Dias, H. Franke, M. Hogstrom, D. Lindquist, G. Pacifici, S. Pappe, B. Rajaraman, J. Rao, R. Ratnaparkhi, R. Smith, and M. Williams, "Software defined environments: An introduction," *IBM Journal of Research and Development*, vol. 58, no. 2/3, pp. 1:1–1:11, March 2014.

[26] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "Nox: Towards an operating system for networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 3, pp. 105–110, Jul. 2008.

[27] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: Enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008.

[28] B. Nunes, M. Mendonca, X.-N. Nguyen, K. Obraczka, and T. Turletti, "A survey of software-defined networking: Past, present, and future of programmable networks," *Communications Surveys Tutorials, IEEE*, vol. 16, no. 3, pp. 1617–1634, Third 2014.

[29] H. Yin, H. Xie, T. Tsou, D. Lopez, P. Aranda, and R. Sidi, "SDNi: A Message Exchange Protocol for Software Defined Networks (SDNS) across Multiple Domains," Internet Draft, Internet Engineering Task Force, June 2012.

[30] D. Kreutz, F. M. V. Ramos, P. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *CoRR*, vol. abs/1406.0440, 2014.

[31] S. Balsamo and A. Marin, "Queueing networks," *Lecture Notes*, pp. 1–81, May 2007.

[32] "Decision Trees," online, Access date: 19 Nov. 2017. [Online]. Available: https://scikit-learn.org/stable/modules/tree.html

[33] W. Rankothge, J. Ma, F. Le, A. Russo, and J. Lobo, "Towards making network function virtualization a cloud computing service," in *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*. IEEE, 2015, pp. 89–97.

[34] J. A. Wickboldt, L. Z. Granville, F. Schneider, D. Dudkowski, and M. Brunner, "Rethinking cloud platforms: Network-aware flexible resource allocation in iaas clouds," in *2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013)*. IEEE, 2013, pp. 450–456.

[35] "Designing Virtual Network Security Architectures," online, Access date: 16 Sept. 2017. [Online]. Available: https://www.rsaconference.com/writable/presentations/file_upload/csv-r03-designing_virtual_network_security_architectures.pdf

[36] SAVI, "Smart applications on virtual infrastructure," http://www.savinetwork.ca/.

[37] "Openstack documentation, http://docs.openstack.org/," online, Access date: 20 Agu. 2015.

[38] G. Wang, T. E. Ng, and A. Shaikh, "Programming your network at run-time for big data applications," in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*. ACM, 2012, pp. 103–108.

[39] M. Veiga Neves, C. De Rose, K. Katrinis, and H. Franke, "Pythia: Faster big data in motion through predictive software-defined network optimization at runtime," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, 2014, pp. 82–90.

[40] N. Handigol, S. Seetharaman, M. Flajslik, N. McKeown, and R. Johari, "Plug-n-serve: Load-balancing web traffic using openflow," Sigcomm Demonstration, 2009.

[41] R. Wang, D. Butnariu, and J. Rexford, "Openflow-based server load balancing gone wild," in *Proceedings of the 11th USENIX Conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*, ser. Hot-ICE'11, 2011, pp. 12–12.

[42] "Mininet Emulator," online, Access date: 20 Jan. 2016. [Online]. Available: http://mininet.org/

[43] D. Turull, M. Hidell, and P. Sjodin, "Performance evaluation of openflow controllers for network virtualization," in *IEEE 15th International Conference on High Performance Switching and Routing (HPSR)*, July 2014, pp. 50–56.

[44] M. Jarschel, S. Oechsner, D. Schlosser, R. Pries, S. Goll, and P. Tran-Gia, "Modeling and performance evaluation of an openflow architecture," in *Teletraffic Congress (ITC), 2011 23rd International*, Sept 2011, pp. 1–7.

[45] L. Yao, P. Hong, and W. Zhou, "Evaluating the controller capacity in software defined networking," in *Computer Communication and Networks (ICCCN), 2014 23rd International Conference on*, Aug 2014, pp. 1–6.

[46] S. Azodolmolky, R. Nejabati, M. Pazouki, P. Wieder, R. Yahyapour, and D. Simeonidou, "An analytical model for software defined networking: A network calculus-based approach," in *IEEE Global Communications Conference (GLOBECOM)*, Dec 2013, pp. 1397–1402.

[47] K. Mahmood, A. Chilwan, and M. Jarschel, "On the modeling of openflow-based sdns: The single node case," *Proceedings of Computer Science and Information Technology*, vol. 4, pp. 207–214, 2014.

[48] Arbor Networks, "Worldwide infrastructure security report," 2015.

[49] C. Barna, M. Shtern, M. Smit, V. Tzerpos, and M. Litoiu, "Mitigating dos attacks using performance model-driven adaptive algorithms," *ACM Trans. Auton. Adapt. Syst.*, vol. 9, no. 1, pp. 3:1–3:26, Mar. 2014.

[50] ——, "Model-based Adaptive DoS Attack Mitigation," in *Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, ser. SEAMS '12. IEEE Press, 2012, pp. 119–128.

[51] C. Barna, M. Shtern, M. Smit, H. Ghanbari, and M. Litoiu, "Model-driven elasticity and dos attack mitigation in cloud environments," in *11th International Conference on Autonomic Computing (ICAC 14).* Philadelphia, PA: USENIX Association, Jun. 2014, pp. 13–24.

[52] A. Kalliola, K. Lee, H. Lee, and T. Aura, "Flooding DDoS Mitigation and Traffic Management with Software Defined Networking," in *Cloud Networking (CloudNet), 2015 IEEE 4th International Conference on*, Oct 2015, pp. 248–254.

[53] B. Wang, Y. Zheng, W. Lou, and Y. Hou, "DDoS Attack Protection in the Era of Cloud Computing and Software-Defined Networking," in *Network Protocols (ICNP), 2014 IEEE 22nd International Conference on*, Oct 2014, pp. 624–629.

[54] C. J. Fung and B. McCormick, "Vguard: A distributed denial of service attack mitigation method using network function virtualization," in *2015 11th International Conference on Network and Service Management (CNSM)*, Nov 2015, pp. 64–70.

[55] R. Han, L. Guo, M. M. Ghanem, and Y. Guo, "Lightweight resource scaling for cloud applications," in *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, May 2012, pp. 644–651.

[56] L. Wu, S. K. Garg, S. Versteeg, and R. Buyya, "Sla-based resource provisioning for hosted software-as-a-service applications in cloud computing environments," *IEEE Transactions on services computing*, vol. 7, no. 3, pp. 465–485, 2014.

[57] G. Fan, H. Yu, and L. Chen, "A formal aspect-oriented method for modeling and analyzing adaptive resource scheduling in cloud computing," *IEEE Transactions on Network and Service Management*, vol. 13, pp. 281–294, 2016.

[58] W. Iqbal, A. Erradi, and A. Mahmood, "Dynamic workload patterns prediction for proactive auto-scaling of web applications," *Journal of Network and Computer Applications*, 2018.

[59] J. Huang, C. Li, and J. Yu, "Resource prediction based on double exponential smoothing in cloud computing," in *2012 2nd International Conference on Consumer Electronics, Communications and Networks (CECNet)*, April 2012, pp. 2056–2060.

[60] R. S. Shariffdeen, D. T. S. P. Munasinghe, H. S. Bhathiya, U. K. J. U. Bandara, and H. M. N. D. Bandara, "Workload and resource aware proactive auto-scaler for paas cloud," in *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*, June 2016, pp. 11–18.

[61] A. Y. Nikravesh, S. A. Ajila, and C. H. Lung, "Towards an autonomic auto-scaling prediction system for cloud resource provisioning," in *2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, May 2015, pp. 35–45.

[62] A. Atrey, G. V. Seghbroeck, B. Volckaert, and F. D. Turck, "Brahma+: A framework for resource scaling of streaming and asap time-varying workflows," *IEEE Transactions on Network and Service Management*, vol. 15, no. 3, pp. 894–908, Sept 2018.

[63] S. Khatua, A. Ghosh, and N. Mukherjee, "Optimizing the utilization of virtual resources in cloud environment," in *2010 IEEE International Conference on Virtual Environments, Human-Computer Interfaces and Measurement Systems*, Sept 2010, pp. 82–87.

[64] A. Bauer, N. Herbst, S. Spinner, A. Ali-eldin, and S. Kounev, "Chameleon: A hybrid, proactive auto-scaling mechanism on a level-playing field," *IEEE Transactions on Parallel and Distributed Systems*, pp. 1–1, 2018.

[65] C. Barna, H. Ghanbari, M. Litoiu, and M. Shtern, "Hogna: A platform for self-adaptive applications in cloud environments," in *Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2015 IEEE/ACM 10th International Symposium on*, May 2015, pp. 83–87.

[66] M. Litoiu and C. Barna, "A performance evaluation framework for web applications," *Journal of Software: Evolution and Process*, vol. 25, no. 8, pp. 871–890, 2013.

[67] L. Baresi, S. Guinea, A. Leva, and G. Quattrocchi, "A discrete-time feedback controller for containerized cloud applications," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. ACM, 2016, pp. 217–228.

[68] W. Iqbal, M. N. Dailey, D. Carrera, and P. Janecek, "Adaptive resource provisioning for read intensive multi-tier applications in the cloud," *Future Gener. Comput. Syst.*, vol. 27, no. 6, pp. 871–879, Jun. 2011.

[69] R. Dobbins, C. Morales, D. Anstee, J. Arruda, T. Bienkowski, M. Hollyman, C. Labovitz, J. Nazario, E. Seo, and R. Shah, "Worldwide Infrastructure Security Report," http://www.arbornetworks.com/dmdocuments/ISR2010_EN.pdf, Arbor Networks, Tech. Rep., 2010.

[70] C. Hoff, "From ddos (distributed denial of service) to edos (economic denial of sustainability)," https://rationalsecurity.typepad.com/blog/2008/11/cloud-computing-security-from-ddos-distributed-denial-of-service-to-edos-economic-denial-of-sustaina.html.

[71] M. Shtern, M. Smit, B. Simmons, and M. Litoiu, "A runtime cloud efficiency software quality metric," in *Companion Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE Companion 2014. ACM, 2014, pp. 416–419.

[72] M. Shtern, R. Sandel, M. Litoiu, C. Bachalo, and V. Theodorou, "Towards mitigation of low and slow application ddos attacks," in *Cloud Engineering (IC2E), 2014 IEEE International Conference on*, March 2014, pp. 604–609.

[73] IBM, "An architectural blueprint for autonomic computing," IBM, Tech. Rep., 2005.

[74] "Open vSwitch," online, Access date: 4 Jan. 2018. [Online]. Available: http://openvswitch.org/

[75] M. F. Bari, S. R. Chowdhury, R. Ahmed, and R. Boutaba, "On orchestrating virtual network functions," in *Proceedings of the 2015 11th International Conference on Network and Service Management (CNSM)*, ser. CNSM '15. IEEE Computer Society, 2015, pp. 50–56.

[76] "Ryu SDN Framework," online, Access date: 16 Sept. 2017. [Online]. Available: http://osrg.github.io/ryu/

[77] "Project floodlight," Online:http://www.projectfloodlight.org/projects/, 2012, access date: April 6,2015.

[78] "RabbitMQ," online, Access date: 28 Dec. 2015. [Online]. Available: http://rabbitmq.com

[79] "Snort," online, Access date: 28 Dec. 2015. [Online]. Available: https://www.snort.org/

[80] "ARP Flux issue in Linux," online, Access date: 19 Nov. 2015. [Online]. Available: http://serverfault.com/questions/336021/two-network-interfaces-and-two-ip-addresses-on-the-same-subnet-in-linux

[81] M. Hüttermann, *DevOps for developers.* Apress, 2012.

[82] F. Gauthier and E. Merlo, "Fast detection of access control vulnerabilities in PHP applications," in *19th Working Conference on Reverse Engineering, WCRE 2012, Kingston, ON, Canada, October 15-18, 2012*, 2012, pp. 247–256.

[83] M. Maggio, H. Hoffmann, A. V. Papadopoulos, J. Panerati, M. D. Santambrogio, A. Agarwal, and A. Leva, "Comparison of decision-making strategies for self-optimization in autonomic computing systems," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 7, no. 4, p. 36, 2012.

[84] M. Emami-Taba and L. Tahvildari, "A Bayesian game decision-making model for uncertain adversary types," in *Proceedings of IBM International Conference on Computer Science and Software Engineering*, 2016, pp. 39–49.

[85] N. Beigi-Mohammadi, C. Barna, M. Shtern, H. Khazaei, and M. Litoiu, "CAAMP: Completely automated DDoS attack mitigation platform in hybrid clouds," in *International Conference of Network and Service Management (CNSM).* IEEE, 2016.

[86] E. Yuan, S. Malek, B. Schmerl, D. Garlan, and J. Gennari, "Architecture-based self-protecting software systems," in *Proceedings of the 9th international ACM Sigsoft conference on Quality of software architectures.* ACM, 2013, pp. 33–42.

[87] M. Salehie, L. Pasquale, I. Omoronyia, R. Ali, and B. Nuseibeh, "Requirements-driven adaptive security: Protecting variable assets at runtime," in *Requirements Engineering Conference (RE), 2012 20th IEEE International*. IEEE, 2012, pp. 111–120.

[88] H. Khazaei, M. Fokaefs, S. Zareian, N. Beigi-Mohammadi, B. Ramprasad, M. Shtern, P. Gaikwad, and M. Litoiu, "How do I choose the right NoSQL solution? a comprehensive theoretical and experimental survey," *Accepted in Journal of Big Data and Information Analytics (BDIA)*, 2016.

[89] P. Zoghi, M. Shtern, M. Litoiu, and H. Ghanbari, "Designing adaptive applications deployed on cloud environments," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 10, no. 4, p. 25, 2016.

[90] M. Smit, M. Shtern, B. Simmons, and M. Litoiu, "Partitioning applications for hybrid and federated clouds," in *Proceedings of the 2012 Conference of the Center for Advanced Studies on Collaborative Research*, ser. CASCON '12. IBM Corp., 2012, pp. 27–41.

[91] K. Xu, K. Tang, R. Bagrodia, M. Gerla, and M. Bereschinsky, "Adaptive bandwidth management and qos provisioning in large scale ad hoc networks," in *Military Communications Conference, 2003. MILCOM'03. 2003 IEEE*, vol. 2. IEEE, 2003, pp. 1018–1023.

[92] M. Welsh and D. E. Culler, "Adaptive overload control for busy internet servers," in *USENIX Symposium on Internet Technologies and Systems*. Seattle, WA, 2003, pp. 4–4.

[93] N. Beigi-Mohammadi, H. Khazaei, M. Shtern, C. Barna, and M. Litoiu, "Adaptive service management for cloud applications using overlay networks," in *15th IFIP/IEEE International Symposium on Integrated Network Management (IM)*, 2017.

[94] ——, "Implementation of self-managing applications on cloud using overlay networks," in *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, May 2017, pp. 871–872.

[95] "VXLAN: A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks," online, Access date: 4 Jan. 2018. [Online]. Available: https://tools.ietf.org/html/draft-mahalingam-dutt-dcops-vxlan-00

[96] N. Beigi-Mohammadi, M. Shtern, and M. Litoiu, "A model-based application autonomic manager with fine granular bandwidth control," in *2017 13th International Conference on Network and Service Management (CNSM)*, Nov 2017, pp. 1–5.

[97] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[98] B. V. V. ZANTEN, "The 9 types of online business models; which one do you use?" https://thenextweb.com/entrepreneur/2011/05/25/the-9-types-of-online-business-models-which-one-do-you-use/, year=2011.

[99] B. UK, "Monetizing your web app: business model options," https://www.boxuk.com/insight/blog-posts/monetizing-your-web-app-business-models, year=2009.

[100] C. Qu, R. N. Calheiros, and R. Buyya, "Auto-scaling web applications in clouds: A taxonomy and survey," *ACM Comput. Surv.*, vol. 51, no. 4, pp. 73:1–73:33, Jul. 2018.

[101] H. Khazaei, J. Mišić, and V. B. Mišić, "Performance of cloud centers with high degree of virtualization under batch task arrivals," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 12, pp. 2429–2438, December 2013.

[102] J. Vilaplana, F. Solsona, I. Teixidó, J. Mateo, F. Abella, and J. Rius, "A queuing theory model for cloud computing," *The Journal of Supercomputing*, vol. 69, no. 1, pp. 492–507, 2014.

[103] H. Khazaei, J. Mišić, and V. B. Mišić, "A fine-grained performance model of cloud computing centers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 11, pp. 2138–2147, November 2013.

[104] H. Khazaei, J. Mišić, V. B. Mišić, and S. Rashwand, "Analysis of a pool management scheme for cloud computing centers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 5, pp. 849–861, 2013.

[105] D. Bruneo, "A stochastic model to investigate data center performance and qos in iaas cloud computing systems," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 25, no. 3, pp. 560–569, 2014.