

# PERFORMANCE-ORIENTED VM PROVISIONING IN CLOUDS

by

Yasir Shoaib

B.A.Sc., University of Toronto, 2008

M.A.Sc., Ryerson University, 2011

A dissertation

presented to Ryerson University

in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

in the program of

Electrical and Computer Engineering

Toronto, Ontario, Canada, 2019

© Yasir Shoaib, 2019

All Rights Reserved

## **Author's Declaration For Electronic Submission Of A Dissertation**

I hereby declare that I am the sole author of this dissertation. This is a true copy of the dissertation, including any required final revisions, as accepted by my examiners.

I authorize Ryerson University to lend this dissertation to other institutions or individuals for the purpose of scholarly research

I further authorize Ryerson University to reproduce this dissertation by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

I understand that my dissertation may be made electronically available to the public.

# PERFORMANCE-ORIENTED VM PROVISIONING IN CLOUDS

Yasir Shoaib

Doctor of Philosophy

Electrical and Computer Engineering

Ryerson University, 2019

## Abstract

Managing applications on the cloud requires extensive decision making on the part of the Application Provider (AP). When an application faces changing workload, the services of the application are either scaled up or down in response. The services run on Virtual Machines (VM) or container instances. APs decide on how the application scales through VM provisioning and the placement of the services on the VMs. Various drivers guide this decision making. Application performance and cost are two such drivers. This thesis answers the question of how APs can meet the performance constraints of their applications while minimizing the cost of the running VMs.

Two versions of the problem are presented. The first version expects to meet mean response time constraints given a deployment configuration through the replication of VMs and addition of virtual processors. The presented solution is based on layered bottlenecks. A case study shows the solution meets response time constraints and uses fewer resources in comparison to a simple utilization based approach.

The second version adds the minimization of cost as an objective, where VM-types having different cost rates are used. This problem does not require a deployment configuration and provides a complete solution, where resources can be added and removed. A novel solution based on the layered bottleneck strength value with genetic algorithm has been presented. For the case study, a decision maker is implemented for a web application. The proposed solution is compared with three algorithms, all of which run within the decision maker. The results from the case study show that the proposed solution provides shorter runtime than the exhaustive search, and is able to meet response time constraints with near optimal minimization of cost. The solution also results in better cost than a plain genetic algorithm and random search, at the expense of slightly longer runtime.

## **Acknowledgments**

First and foremost, I thank God for guiding me and allowing me to successfully finish the task that I started work on years ago.

The insight and feedback of my supervisor, Dr. Olivia Das, has been quite valuable in the progression and completion of this thesis. I thank her for her mentorship and cooperation.

I thank Ryerson University for providing me the facilities to perform my research. To Government of Ontario and Ryerson University, I am grateful for the various scholarships and awards they have honoured me with.

Many thanks to the members of my thesis committee, who provided feedback on my work and suggested improvements that allowed me to enhance the quality of my research work and the thesis. Also, I thank all the reviewers of my published works for their feedback and valuable time.

I appreciate the keen interest shown by my co-workers on my research topic, and for their thought provoking questions and suggestions. Thanks to Dr. Prakash Atreya for advising me to present my ideas. Also, I would like to extend my thanks to Savita Pahuja and Yuri Litvinovich for their feedback. My sincere thanks to Tatjana Papic for attending my thesis defence and for her support.

To my family and friends, who have keenly asked throughout the years about the progress of my thesis, thank you for your kind words of encouragement and support.

My family has supported my efforts throughout the years and have been a part of my journey. Many thanks to my parents, parents-in-law and siblings for their encouragement. And, thank you to my wife, Arfa, for her love, cooperation and support. Special thanks to my bundle of joy, Amaan, my carbon copy, for the beautiful gift you are, for giving me happiness and joy, day-in and day-out.



## Dedication

*To my family*

$$6a + h + 2n + s + k + u$$

# Contents

<b>Author's Declaration</b>	<b>ii</b>
<b>Abstract</b>	<b>iii</b>
<b>Acknowledgements</b>	<b>iv</b>
<b>Dedication</b>	<b>v</b>
<b>List of Tables</b>	<b>xii</b>
<b>List of Figures</b>	<b>xiv</b>
<b>Listings</b>	<b>xv</b>
<b>List of Algorithms</b>	<b>xvi</b>
<b>List of Appendices</b>	<b>xvii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Cloud Dynamic Provisioning . . . . .	2
1.2 Related work . . . . .	2
1.3 Research Overview . . . . .	4
1.4 Contributions . . . . .	5
1.5 Thesis outline . . . . .	6
<b>2 Background</b>	<b>7</b>
2.1 Cloud Computing . . . . .	7
2.2 Virtualization . . . . .	8

2.3	Performance Evaluation . . . . .	8
2.3.1	Queueing Networks . . . . .	10
2.3.2	Layered Queueing Networks (LQN) . . . . .	11
2.3.3	Layered Bottlenecks . . . . .	14
2.4	Genetic Algorithm . . . . .	15
2.4.1	Initialization . . . . .	16
2.4.2	Encoding . . . . .	16
2.4.3	Evaluation and the fitness function . . . . .	17
2.4.4	Selection operation . . . . .	17
2.4.5	Crossover operation . . . . .	17
2.4.6	Mutation operation . . . . .	17
2.4.7	Termination criteria . . . . .	18
2.5	Conclusions . . . . .	18
<b>3</b>	<b>Cloud Provisioning for Meeting Performance Goals: Taxonomy and Survey</b>	<b>19</b>
3.1	Classifying cloud provisioning . . . . .	21
3.2	Who does cloud provisioning? . . . . .	22
3.3	Which provisioning types, scaling, problems and placement schemes exist? . . . . .	25
3.3.1	Provisioning Problems . . . . .	25
3.3.2	Provisioning types . . . . .	26
3.3.3	Horizontal and Vertical scaling . . . . .	28
3.3.4	Placement schemes . . . . .	28
3.4	When is the provisioning decision made? . . . . .	30
3.5	What resources are provisioned? . . . . .	31
3.6	Where are the resources deployed? . . . . .	32
3.7	How are the provisioning problems solved? . . . . .	34
3.7.1	Provisioning policies . . . . .	34
3.7.2	Algorithms . . . . .	38
3.7.3	Techniques and Technologies . . . . .	39
3.8	Toward performance-oriented cloud provisioning . . . . .	39

3.9	Observations	45
3.10	Conclusions	46
<b>4</b>	<b>VM provisioning using layered bottlenecks</b>	<b>47</b>
4.1	The Provisioning Problem	47
4.1.1	Assumptions	48
4.1.2	Notations	49
4.1.3	Formal problem definition	50
4.2	The Solution	51
4.2.1	Layered bottlenecks solution	51
4.2.2	jLQNInterface Java library	52
4.2.3	Contributions Overview	52
4.2.4	Provisioning framework	53
4.2.5	Controller Algorithm	54
4.3	Case Study	58
4.4	Conclusions	61
<b>5</b>	<b>The Decision Maker</b>	<b>63</b>
5.1	The provisioning problem	63
5.1.1	Assumptions	64
5.1.2	Notations	67
5.1.3	Formal problem definition	69
5.2	The solution	69
5.2.1	The decision maker	70
5.2.2	Dynamic Provisioning Control System	71
5.2.3	The decision making process	73
5.2.4	Problem Encoding	74
5.2.5	Number of Evaluations	75
5.2.6	Algorithm parameters	75
5.3	Decision Maker Algorithms	76
5.3.1	Exhaustive Search Algorithm (ESA)	76

5.3.2	Random Search Algorithm (RSA)	76
5.3.3	Simple Genetic Algorithm for Performance (SGAP)	77
5.3.4	Performance-oriented Genetic Algorithm (PGA)	78
5.4	Discussions	83
5.4.1	Similarities and Differences: SatisfyQoS and PGA	84
5.4.2	Choosing genetic algorithm	84
5.4.3	Limitations	85
5.4.4	Architecture	85
5.4.5	Handling failures	87
5.4.6	Timeliness	88
5.4.7	Hybrid policy	88
5.4.8	Monitoring and alerting	88
5.4.9	Deploying new versions of the decision maker	89
5.4.10	Enforcing cost and security of the decision maker	89
5.4.11	Performance	89
5.5	Conclusions	90
<b>6</b>	<b>Case study: Bike routes application</b>	<b>91</b>
6.1	Bike routes web application	91
6.1.1	Overview	91
6.1.2	LQN Performance Model	92
6.2	Decision Maker Inputs	97
6.3	Results and analysis	99
6.3.1	Impact of parameters	99
6.3.2	An optimal configuration	101
6.3.3	Cost vs. Iterations - Optimality Evaluation	101
6.3.4	Cost vs. Users	102
6.3.5	Runtime vs Users	105
6.4	Conclusions	106
<b>7</b>	<b>Conclusions</b>	<b>108</b>

7.1	Contributions . . . . .	108
7.1.1	SatisfyQoS . . . . .	108
7.1.2	PGA . . . . .	109
7.2	Limitations . . . . .	110
7.3	Future work . . . . .	110
7.3.1	Multi-cloud . . . . .	110
7.3.2	Minimal change . . . . .	111
7.3.3	Adding Cost constraint . . . . .	111
7.3.4	Discrete-Event-Simulation . . . . .	111
	<b>Appendices</b>	<b>112</b>
A	<b>Permissions to use copyrighted material</b>	<b>113</b>
A.1	IEEE Permissions - UCC 2012 paper . . . . .	113
A.2	ENTCS permission - 2011 paper . . . . .	114
B	<b>Software architecture: The Decision Maker</b>	<b>115</b>
B.1	Requirements and Design decisions . . . . .	115
B.2	Software components . . . . .	116
B.2.1	Maven projects . . . . .	117
B.2.2	Semantic versioning . . . . .	119
B.2.3	Testing . . . . .	119
B.2.4	Version control . . . . .	119
B.3	jLQNInterface library . . . . .	119
B.3.1	Example . . . . .	120
B.3.2	Classes . . . . .	123
B.4	application-cloud-model library . . . . .	124
B.4.1	Example . . . . .	124
B.4.2	Class diagram . . . . .	128
B.5	The decision maker software . . . . .	129
B.5.1	Inputs . . . . .	129

B.5.2 Class diagram . . . . .	130
B.6 MOEA Framework . . . . .	131
B.7 Conclusions . . . . .	131
<b>C Results</b>	<b>132</b>
C.1 Data and graphs . . . . .	132
<b>Bibliography</b>	<b>142</b>
<b>Acronyms</b>	<b>159</b>
<b>Glossary</b>	<b>161</b>
<b>Index</b>	<b>170</b>

# List of Tables

3.1	Dynamic provisioning and associated Problems . . . . .	26
3.2	Placement schemes . . . . .	29
4.1	Summary: Case 1 and Case 2 results . . . . .	61
6.1	Service Demand Parameters for Base-Scenario Model . . . . .	95
6.2	VM Types . . . . .	97
6.3	Parameters . . . . .	98
6.4	Case Study: Hardware and Software Configuration . . . . .	98
6.5	Selected values for the Parameters . . . . .	99
6.6	Number of evaluations for the algorithms . . . . .	105



# List of Figures

2.1	An LQN model . . . . .	12
3.1	Cloud Provisioning - Classification . . . . .	20
3.2	Cloud control hierarchy IaaS . . . . .	23
3.3	Cloud control hierarchy CaaS . . . . .	24
3.4	Cloud Provisioning . . . . .	27
3.5	Multiple Clouds . . . . .	33
4.1	Cloud Provisioning . . . . .	48
4.2	Provisioning framework: Single step process flow diagram . . . . .	53
4.3	Provisioning framework: Inter-step process flow diagram . . . . .	53
4.4	Case study input model . . . . .	58
5.1	Cloud Provisioning . . . . .	63
5.2	Load Balancing and VM replication . . . . .	65
5.3	Decision maker block diagram . . . . .	70
5.4	How a generic dynamic provisioning control system works . . . . .	71
5.5	The decision making process . . . . .	74
5.6	LBX when both parents meet response time constraints . . . . .	81
6.1	Base-Scenario LQN Model with no input parameters . . . . .	93
6.2	Case Study LQN Model . . . . .	96
6.3	Impact of Parameters - PGA . . . . .	100
6.4	Result of 220 users for ESA - Case Study LQN Model . . . . .	101
6.5	Cost vs. Iterations (Users=20) . . . . .	102
6.6	Cost vs. Iterations (Users=220) . . . . .	103

6.7 Cost vs. Users . . . . . 104

6.8 Cost vs. MaxVmsPerTask (Users=20 and 220) . . . . . 105

6.9 Runtime vs. Users . . . . . 106

# Listings

4.1 Case 1 result based on Layered Bottlenecks . . . . .	59
4.2 Case 2 result based on resource saturation . . . . .	60

# List of Algorithms

2.1	SIMPLEGENETICALGORITHM (SGA)	16
4.1	Controller SATISFYQoS	55
4.2	FINDBOTTLENECKS	57
5.1	RANDOM SEARCH ALGORITHM (RGA)	76
5.2	LAYERED BOTTLENECK CROSSOVER (LBX)	79
5.3	LAYERED BOTTLENECK MUTATION (LBM)	82

# List of Appendices

A	Permissions to use copyrighted material . . . . .	113
B	Software architecture: The Decision Maker . . . . .	115
C	Results . . . . .	132

# Chapter 1

## Introduction

Soaring number of Internet users and the day-to-day fluctuations in their use of the applications on the web, compels the [Application Provider \(AP\)](#) to focus on the capacity and performance of their applications. When larger than expected number of users access the applications, they contribute to increasing the workload on the applications. This stresses the [AP](#) resources and systems and eventually causes service degradation.

As demand for application services increase, there is a continuous need to re-architect infrastructure and provision adequate infrastructure capacity. This is why, cloud offerings have become quite attractive for large scale operations; especially when scaling needs to happen quickly and on-demand. This aids in both provisioning capacity and improving performance.

From January 2017 to January 2018, the number of Internet users have risen by 7%, totalling over 4 billion in 2018 [1]. Social media users are now over 3 billion in number [1]. This increase in customers of Internet facing applications will propel application providing organizations to consider cloud offerings for capacity and performance reasons, if they haven't already done so.

Along with performance benefits from quick and on-demand resource allocations, cost minimization is another factor driving the move to the cloud. With cloud computing's utility-based pricing model, the customers pay for the resources used; in comparison, they would have incurred capital expenses if resources were purchased [2].

Considering these reasons, this thesis focuses on cost and performance of applications that run on clouds.

**Chapter Outline** Section 1.1 explains how dynamic provisioning within cloud systems is used for adequately provisioning resources to meet different objectives. The end of the section presents the

thesis statements. Section 1.2 presents the related works. Section 1.3 provides an overview of the research in the thesis. Section 1.4 outlines the contributions of this thesis. Section 1.5 presents the thesis outline.

## 1.1 Cloud Dynamic Provisioning

Interest in cloud computing has soared rapidly in the recent years. With the adoption and move to cloud computing, the focus of industry and academia has moved toward the challenges facing this computing paradigm. One of such challenges is that of autonomously provisioning adequate resources by automatically adding and removing them to handle the fluctuating Internet user request demands [3,4]. This is a problem known as “dynamic provisioning” [3], “dynamic resource provisioning” [4] or [autoscaling](#).

Under-provisioning of resources, i.e. having fewer resources allocated for serving the user workload, cause web application [end-users](#) to experience excessive delays, especially during demand surges. Eventually, due to poor performance, disgruntled users leave the site, incurring loss to the [AP](#) businesses — as seen happening with eCommerce sites [5]. On the other hand, over-provisioning leads to higher costs for the [Cloud Provider \(CP\)](#) due to management of large number of servers that because of being under-utilized cause excess power consumption and heat dissipation in their data centers. In this case, the loss is not only restricted to CP but also extends to AP, who have constrained budgets but have to pay for unnecessary containers or Virtual Machines (VM). Therefore, the main problem arises owing to either resource under-provisioning, where resources are over-utilized and lead to poor performance, or resource over-provisioning, where resources are under-utilized and waste energy. The quintessential scenario would be to have dynamic provisioning of resources through the AP, following the fluctuating workload demands of the application, to meet the response time constraints and minimize cost, which in practise is quite a challenging endeavour.

## 1.2 Related work

A core component of a dynamic provisioning system is the [decision maker](#). Various policies have been adopted within the decision maker for making provisioning decisions. Rule-based policies are

quite common in industry (e.g. Amazon EC2 Dynamic Scaling [6], and Microsoft Azure Autoscale [7]). In these policies, the approach is to perform actions such as allocation and de-allocation of resources based on triggered events such as resource utilizations exceeding or falling below certain thresholds, respectively. Qu et al. [8] in their survey, point to the following shortcomings of simple rule based policies:

- these policies warrant the need to properly understand the application behaviour to correctly set the thresholds and corresponding actions
- these policies are incapable of adjusting to fluctuating application workload

Along with rule-based policies, schedule-based autoscaling is also offered through Amazon EC2 Schedule Scaling [6], and Microsoft Azure Autoscale [7]. Schedule based policies are affected by the same shortcomings of the rule based approach.

Another popular decision making approach is machine learning, where through learning, a dynamic resource provisioning plan is determined over time (e.g. [9,10,11]). This approach suffers from poor “initial performance” [9]. These models require time to converge and stabilize, causing the “auto-scaler to perform poorly during the active learning period” [8]. Also, the learning time is difficult to determine [8].

Analytical modeling overcomes the limitations posed by rule-based policies [9]. The analytical modeling approach also does not see the learning challenges faced by machine learning. In the domain of dynamic provisioning, analytical performance models are often built on queueing theory, where network of queues — Queueing Networks (QN) — become essential for modeling multi-tier applications to describe their [application architecture](#) [8].

With regards to provisioning and modeling of multi-tier applications, the easier approach is to split the overall metric (e.g. response time), to the metric of each service or each tier, and solve the problem (e.g. [12, 13, 14]), but this “results in suboptimal solutions globally” [8]. The more challenging approach is to consider the metric for the whole application and find the optimal deployment configuration “holistically” [8].

Our work [15] (Chapter 4), has employed analytical modeling for dynamic provisioning to meet performance constraints of multi-tier applications by identifying “layered bottlenecks” [16], and based on the identified bottleneck, scaling either VMs or virtual processors. In the work, [SatisfyQoS](#)



algorithm is presented for directing provisioning decisions, which through a case study is shown to meet response time constraints using fewer resources in comparison to a simple utilization based approach. Similar to our work, [17] has also proposed dynamic provisioning for multi-tier applications; however, they have used QNs, whereas we employ [Layered Queueing Network \(LQN\)](#) analytical performance models [18] and use them for performance analysis and for identifying layered bottlenecks. LQNs are ideal for representing the interactions and intricacies of multi-tier applications. In case of Remote Procedure Calls (RPC), the queueing that occurs at lower layers due to software contention is included in the upper layer response time of an LQN model [18]. Furthermore, they can also model nested RPCs [19]. LQN models have been used widely in many performance studies (e.g. [20, 21, 22, 23]). Shoaib and Das [22] have detailed the various studies on use of LQN modeling for studying of software and web systems.

Works on multi-tier applications that find an optimal configuration holistically (e.g. [15, 17]), begin with a given deployment configuration and result in servers being added or removed from the most or least utilized tier until the estimated application performance is acceptable for the given workload. However, when the difference between current deployment configuration and the future state of the configuration is large, this strategy of searching through the neighbours and performing incremental change in the number of servers, takes a long time to find the most suitable configuration. To add, these works do not consider variation of computation power and cost. Along with these challenges, “the search of the optimal provisioning plan with a combination of heterogeneous VMs is often computing intensive” [8]. Thus, finding an optimal configuration for a multi-tiered application that minimizes the resource cost, meets the performance constraints, and also considers multiple VM types with varying computing power and costs, is a significant research challenge, which has been undertaken for this thesis.

### 1.3 Research Overview

For an application running within a cloud, a [deployment configuration](#) consists of the number and types of VMs (or containers) allocated to the components of the application. The focus of this thesis is on novel search algorithms that can be harnessed by the [AP](#) to find the optimal [deployment configuration](#) of a multi-tiered application, for meeting the performance guarantees

of the application when facing fluctuating user workload, with the objective to minimize cost for the AP.

The novelty of this thesis lies in solving the dynamic provisioning problem for an AP by considering the following four key features together:

1. provisioning for multi-tier applications to find an optimal configuration holistically
2. employing an optimization technique with analytical model-based decision making
3. minimizing cost and meeting the performance constraints
4. considering VM heterogeneity

**Thesis statements** A cloud provisioning problem from the perspective of an AP, using different VM-types with an objective to minimize cost of the application given response time constraints has been formulated in this thesis. Through a case study this thesis demonstrates that the new algorithm, [Performance-oriented Genetic Algorithm \(PGA\)](#), is able to solve this problem by generating feasible and near-optimal solutions. In this thesis, [LQN](#) analytical performance modeling and [Genetic Algorithm \(GA\)](#) are used together for guiding the provisioning decision making. Two new genetic operators are proposed as part of a new provisioning genetic algorithm to solve the dynamic provisioning problem.

## 1.4 Contributions

Following are the contributions of this thesis:

1. [SatisfyQoS](#), a layered bottleneck dynamic provisioning algorithm, which expects to meet mean response time constraints given a [deployment configuration](#) through the replication of VMs and addition of virtual processors has been proposed in this thesis. An implementation of [SatisfyQoS](#) algorithm for a case study, where the solution meets response time constraints and uses fewer resources in comparison to a simple utilization based approach has been presented (Chapter 4)
2. [PGA](#), a new algorithm based on genetic algorithm, where the objective is to minimize cost and meet response time constraints has also been proposed (Chapter 5). Two new genetic operators, [Layered Bottleneck Crossover \(LBX\)](#) and [Layered Bottleneck Mutation \(LBM\)](#)

form the main components of the [PGA](#) (Chapter 5). To evaluate [PGA](#), a case study has been conducted on a web application, where [PGA](#) is compared with [Exhaustive Search Algorithm \(ESA\)](#), [Random Search Algorithm \(RSA\)](#) and [Simple Genetic Algorithm for Performance \(SGAP\)](#). The results show that [PGA](#) outperforms [RSA](#), and [SGAP](#) in decision making (Chapter 6).

## 1.5 Thesis outline

This thesis is organized as follows: Chapter 2 provides a short background of cloud computing, virtualization, performance evaluation, and genetic algorithms. Chapter 3 is a detailed literature survey on dynamic provisioning in cloud systems. Chapter 4 presents the initial research approach to the VM provisioning problem. Chapter 5 presents a complete dynamic provisioning problem and solution to meet performance constraints and minimize cost. Chapter 6 presents a case study on a bike routes web application by using the decision maker. Chapter 7 is the conclusions.

# Chapter 2

## Background

This chapter gives a short background to readers about cloud computing, virtualization, performance evaluation, and genetic algorithms.

**Chapter Outline** Section 2.1 describes the services and the deployment models of cloud systems. Virtualization technology, which makes possible dynamic provisioning within clouds, is mentioned in Section 2.2. Following this, a short detail of performance evaluation is presented in Section 2.3. Performance evaluation has been used in many works to evaluate system performance and used in provisioning solutions. Section 2.4 explains the genetic algorithm. Section 2.5 are the conclusions.

### 2.1 Cloud Computing

For [APs](#), who are developing and deploying web applications, the cloud paradigm facilitates the use of computing resources from public [CPs](#) with “multi-dimensional ease” [15]. [APs](#) strive for deploying web applications that meet high performance standards even when workload demands are at their peak. They meet these request demands either through the management of their own web-server farms or through the purchase of hardware and software as services from cloud computing service providers. With the ability to allow for quick resource addition and removal — a feature known as elasticity [24] — the cloud systems provide a unique capability to their customers [25, 26]. The resources, viz. processors, network, software, etc., remain in CP data centers and can be added and removed as needed by the customer [2, 24]. With cloud computing’s utility-based pricing model, the customers pay for the resources used; in comparison, they would have incurred capital expenses if resources were purchased [2]. These customers considering their spatial, temporal and monetary constraints, could thereby rely on services the cloud is known to offer. The

different levels of cloud services offered by CP are: [Infrastructure-as-a-Service \(IaaS\)](#), [Container-as-a-Service \(CaaS\)](#), [Platform-as-a-Service \(PaaS\)](#), and [Software-as-a-Service \(SaaS\)](#). Well-known providers include Amazon, Google and Microsoft [3, 4]; although the levels of services offered may differ from one provider to another. Furthermore, with various service and deployment models to choose from, such as those models mentioned in “The NIST definition of cloud computing” [24], a customer can either hand-pick desired services from CP or themselves setup a private cloud infrastructure for their own use.

Clouds may be deployed as a public-cloud, community-cloud, private-cloud or hybrid cloud [24]. Public-clouds are provided by organizations such as Amazon, Google and Microsoft who host hardware and software resources in their facilities. Private clouds are hosted by individual organizations to serve their own internal users, whereas community-clouds are hosted for users in a group of organizations [24]. Hybrid clouds are composed as a combination of previously mentioned deployment models [24].

## 2.2 Virtualization

Virtualization technology offers emulated hardware for guest OSs to run on, where each OS runs as a separate VM instance. A virtual machine monitor (VMM), a.k.a. hypervisor is positioned between the host hardware and one or more guest OSs, and manages access of shared resources needed by the guests [27, 28, 29]. The VM instance is an isolated platform and runs on top of a hypervisor. One or more OSs execute under the control of the hypervisor, where each OS is encapsulated within its own VM interacting with the hardware indirectly through the hypervisor [30, 31]. When hypervisor executes right above the physical hardware, the arrangement is known as Type-I virtual environment [32]. If the hypervisor executes within a host OS then such an environment is known as Type-II [32]. Cloud computing is built on top of virtualization technology, which is a “core” [33] technology of the cloud.

## 2.3 Performance Evaluation

The complexity and nature of web applications require their development to follow a Software Development Life Cycle (SDLC) [22]. As users expect web applications to be quick and responsive,

attention to system's performance is paid from an early software development stage and followed throughout the SDLC by means of a well-defined systematic process: Software Performance Engineering (SPE) [34]. As part of the [Software Performance Engineering \(SPE\)](#) process, measurements and performance modeling can be used to study a system. To directly assess if an application will meet the required performance objectives based on available resources, for the purposes of capacity planning [35], a measurement-based approach is adopted. The behaviour of the system under given customer workload can provide results which can help identify performance bottlenecks [36].

Measurement is a direct assessment of the actual or representative system under varying workloads, providing the most accurate results [37]. Simulation and analytical modeling are performance modeling approaches. Performance modeling encapsulates the performance characteristics of a system that are available from its design in a performance model. In simulation modeling, a software representation of the system and its interactions are developed, and statistics collected from the model software's execution provide the performance metrics [38,39]. Analytical modeling requires the use of mathematical equations for their solution.

Analytical models exhibit lower accuracy in comparison to other evaluation methods. However, these are quicker to create and solve, and alongside are easier to manage, to find their usefulness throughout the development life of the software. They are especially useful when a quick performance feasibility evaluation of a conceptual system is desired [37].

Performance modeling, which uses performance models, also finds use in capacity planning by means of predicting system's performance and by pinpointing system bottlenecks. Other uses of modeling include “dynamic capacity provisioning” [40] (a.k.a. “dynamic provisioning” [3,40]) — i.e. allocating and preparing resources to handle the demands — and finding deployment configuration parameters that meet the desired objectives [40].

For evaluating performance, response time and throughput metrics are used. [Response time](#) is the total roundtrip time for a request to complete processing through the system [37]. [Throughput](#) is the rate of customer request completion [37].

For analytical modeling, use of well known Queueing Network (QN) models is very common. In the following subsections a brief description of QN, Layered Queueing Networks (LQN), and layered bottlenecks are provided.

### 2.3.1 Queueing Networks

Behaviour of Client-Server system, Web Applications, Operating System (OS) Kernels, etc. can be represented through Queueing Networks (QNs). Bolch et al. [41] presents case-studies that show the use of Queueing Systems in modeling of the aforementioned systems. QNs are an interconnection of service stations or queues that serve customers or jobs. Depending on the how customers enter and remain in the system, QNs are classified as open, closed or mixed. When new customers enter the system at a defined arrival rate, the system is Open. Systems with fixed number of customers are Closed Queueing Networks.

Customers are served based on the Scheduling Discipline of the service station. If the station is busy with processing a customer, the new arriving customers wait and form a queue. The queue could infinite or of finite capacity. Few of the regularly used scheduling disciplines are [41]:

**FCFS (First-Come-First-Served)** : Customers that arrive first are served before others.

**LCFS (Last-Come-First-Served)** : Last arriving customers are served first.

**PS (Processor Sharing)** : Server processing is shared amongst customers who are given very small execution time slices, as jobs appear to be processed simultaneously.

**Infinite Server** : Also known as Delay Servers. There is no queueing at these servers.

Input to QNs include description of the queueing stations, customer workload intensity and the service demands of the customers [42]. Alongside, the number of stations and scheduling discipline of each station are part of input. Customer workload intensity can either be expressed as the number of concurrent customers in the system (closed-system) or as the arrival rate (open-system). Along with concurrent customers, the think time of the customers may also be specified. Think time, defines the duration of time that the customer waits and thinks before again requesting service from the system. The service demands specify the service time of the customer at each station and the number of visits made to the station.

QNs have single or multiple customer classes. In single-class, customers have the same customer information detail: workload intensity and service demands at the stations. In multi-class, multiple classes of customers exist where the classes differ in their customer information detail.

However, basic QN models fall short in being able to account for software contention as seen in software servers [43]. If effects of queueing due to software contention are ignored then response times are understated leading to inaccurate results. QNs depict software as customers only, whereas software servers behave at times as servers when they are serving their client requests and behave other times as clients themselves when requesting service from other servers, a behaviour which QNs cannot depict [43]. Furthermore, aspects such as parallel software execution that is seen through creation of a child process from an existing process (fork) are difficult to represent [43].

### 2.3.2 Layered Queueing Networks (LQN)

LQN [18] analytical models are based on extended QNs and are designed to eliminate the aforementioned shortcomings of QN. In case of Remote Procedure Calls (RPC), the queueing that occurs at lower layers due to software contention is included in the upper layer response time of an LQN model [18]. Furthermore, they can also model nested RPCs [19]. The parallel execution of software and servers which send “early reply” [19] (i.e. some processing at the server is to happen in second phase after reply is sent to the client) can also be modeled [43]. They are well-suited to depict both complex software applications and the hardware resources that these software entities run on [18]. With the ability to incorporate varying degrees of details in the model, LQN performance modeling can easily be integrated with the SDLC. LQNs are ideal for representing the interactions and intricacies of multi-tier applications and this work therefore uses LQN for performance modeling.

LQN models have been used widely in many performance studies (e.g. [20,21,22,23]). Modeled entities include hardware resources as processors, software processes as tasks, service classes as entries, and finer operation steps of software as activities. Entities have associated multiplicity, i.e. multiple entities with one queue. Processors — and thereby associated resources — can be replicated with each copy having their own queue. These models are able to depict nested interactions between servers considering simultaneous resource possession, and provide results such as mean response time, throughput and utilization of modeled entities [18]. Models could be represented as plain LQN or as XML-based LQN (LQNX), where LQN Solver (LQNS) tool [18] is used for analytically solving these models. For solution purposes, the input LQN model is subdivided into “MVA submodels” [19], which are subsequently solved through approximate mean-value



analysis (MVA) until the results of each layer have converged.

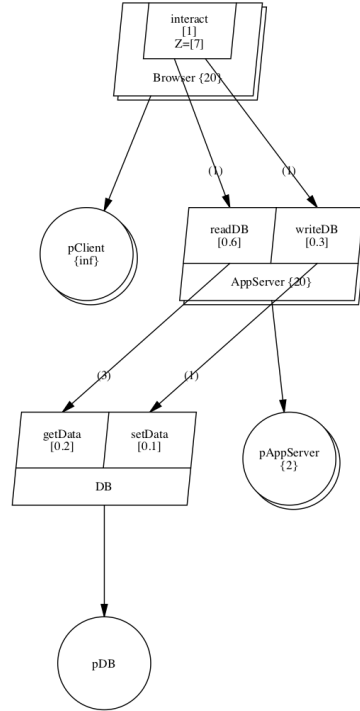


Figure 2.1: An LQN model

**Tasks** An LQN model is shown in Figure 2.1. There are three resources: Browser, AppServer and DB. In LQN modeling, these resources are known as **tasks** and represented by the outer parallelograms. “Resources include, but are not limited to: actual tasks (or processes) in a computer system, customers, buffers, and hardware devices” [18]. Each task has a queue to receive requests. Tasks can be reference tasks, i.e. they only initiate requests but do not receive any. Browser is a reference task.

**Entries** The parallelogram within the tasks correspond to the operations that they perform referred as **entries** [44]. Entries are akin to customer classes of QNs [19].

**Processors** Each task executes on a **processor**, shown as oval shapes with connection to the task. In LQN modeling, processors are entities on which time is used [18]. Each processor has its own

queue.

**Multiplicity** Multiplicity is used in LQN to add tasks and processors. Multiplicity adds multiple queueing servers for handling requests of one queue. In a software context, task multiplicity is used to model multiple threads of a software process. For processors, multiplicity is used to model a multiprocessor. Multiplicity is shown as braces in the figure.

**Communication** Communication between the entries of tasks can be of three types: Synchronous, Asynchronous and Forwarding. In synchronous communication, the requesting task (client) blocks until a response is received from a server task. Phase one of the server begins when it receives a request. After sending a response back to the client, the phase two execution of the server begins. In asynchronous interaction the client task does not block after sending a request. In forwarding communication, the server (task A) that received the request forwards the request to another task (task B). Task A at this point starts execution in phase two and when task B is done processing its request, it sends a response back to the client after which task B will begin its execution in phase two. Nesting of different communication patterns can also happen. In Figure 2.1, synchronous calls are shown. In this thesis we have used synchronous calls.

**The architecture** Figure 2.1 represents a system where there are 20 Browsers sending requests to a dual-processor, 20-threaded AppServer process. The data is stored in the DB database and data manipulation from the Browser happens through readDB and writeDB operations on the AppServer. The Browser has a think time of 7 seconds ( $Z = 7$ ). Each entry has an associated service time per phase. The service times are shown within square brackets. The arrows show the direction in which the requests are made and the parentheses indicate the number of requests made. Unless specified otherwise, the frequency of interactions is one.

**Inputs** The LQN model structure comprises of tasks, entries, host hardware resource of the former components and their interconnections. Inputs to the model are scheduling discipline of the hardware resources, customer workload intensity and the service demands of the customers for the model components at each phase.

**Workflow** In the example, the Browser initiates the requests performing the interact operation, requesting readDB and writeDB operation on the AppServer. To fulfill the readDB request, the AppServer first parses the request and then makes three requests to the DB through the getData entry. After receiving the data response from the DB database the AppServer sends the response back to the Browser, completing the readDB interaction initiated by the Browser. The request from interact to writeDB operation follows a similar workflow. Once the responses have been received from the AppServer, the Client thinks for 7 seconds and then initiates another set of requests to AppServer for readDB and writeDB operations, repeating this process infinitely.

### 2.3.3 Layered Bottlenecks

“Layered bottlenecks” [16] are software or hardware resources that are *saturated* —i.e. they have high utilization — and which hinder performance by constraining the system throughput, as seen occurring in systems with simultaneous resource possession. Until the bottlenecks are eased performance improvements would be unnoticeable. To alleviate the bottlenecks in a system of client-server computing nodes, the options include adding software threads, increasing processors and adding new nodes (replication) of the bottleneck [23]. Specifically for VMs, replication implies instantiating copy VMs (replicas) where the incoming requests would now be equally distributed amongst the replicas.

*Hardware/processor bottlenecks* are easily identified by high utilization of the devices, however, software bottlenecks require a more detailed analysis. *Software bottlenecks* are encountered when hardware is not saturated but the system’s performance is still restricted, e.g. task bottlenecks [16]. The challenge is: not only is the particular software bottleneck saturated but other resources requesting services from it are also saturated because of “pushback” [16]. Furthermore, the resources that the bottleneck depends on by requesting services are unsaturated [16]. Attributes mentioned above are key toward pinpointing these bottlenecks — a process which is detailed below.

Based on the paper by Frank et al. [16] we briefly describe the layered bottleneck algorithm here. Given the performance model and a saturation threshold ( $sat_{thresh}$ ), the algorithm returns the bottleneck. The *saturation* of processors (Equation 2.4) and the **BStrength** (bottleneck strength) (Equation 2.5) of all other resources — which are not processors (e.g. tasks) — are calculated. The processor with the highest saturation is the bottleneck if its saturation is greater than the

threshold. Otherwise, if no processor is a bottleneck, then the resource with the highest **BStrength** value, having its saturation greater than the threshold, is the bottleneck.

Equations useful for finding **layered bottlenecks** are [16]:

$$U_t = \text{utilization of resource } t \quad (2.1)$$

$$m_t = \text{multiplicity of resource } t \quad (2.2)$$

$$sat_t = \text{saturation of resource } t \quad (2.3)$$

$$sat_t = \frac{U_t}{m_t} \quad (2.4)$$

$$BStength_t = \frac{sat_t}{sat_{shadow(T)}} \quad (2.5)$$

$$shadow(T) = \arg \max_{t \text{ in } Below(T)} sat_t \quad (2.6)$$

$$Below(T) = \text{the set of resources that } T \text{ depends on (directly or indirectly)} \quad (2.7)$$

## 2.4 Genetic Algorithm

**GA** is a search algorithm, based on evolutionary theory, where the goal is to solve an optimization problem. In their book, “Artificial Intelligence - A Modern Approach,” Russell and Norving while explaining about a group of algorithms which include genetic algorithm, mention that “[t]hese algorithms are suitable for problems in which all that matters is the solution state, not the path cost to reach it” [45]. This is in contrast to search algorithms such as Dijkstra and A\* where the steps followed to reach the solution state matter. This is because for both Dijkstra and A\* each link has an associated cost and even if the goal state is reached, the optimal solution is the one where the cost of the total path is minimal.

For introducing GAs, researchers give credit to John Holland [46, 47]. It was in his book “Adaptation in Natural and Artificial Systems” [48], published in 1975, when the term “genetic algorithm” was first introduced by Holland [46].

Over the years, lot of effort has been put forward in understanding and applying GAs. Mehboob et al. [49] in their survey have provided various reasons for using GAs. They mention: “GAs are

---

**Algorithm 2.1** SIMPLEGENETICALGORITHM (SGA). Based on Srinivas & Patnaik [47]

---

**Input:** Algorithm parameters

**Output:** Fittest individual from the last generation

```
1: INITIALIZATION()
2: EVALUATION()
3: while termination-criteria-not-satisfied do
4:   SELECTION()
5:   Crossover()
6:   MUTATION()
7:   EVALUATION()
8: end while
```

---

very good at navigating through huge search spaces to heuristically find near-optimal solutions in quick time” [49].

Genetic algorithm follows a natural selection process [45]. In this algorithm, generations from an initial population are created through reproduction, where the fittest individuals have a higher probability of being selected to produce children. Through this selection process the generations continue to converge toward the solution, to find individual(s) that solve the problem.

There are many variants of the genetic algorithm. The “Simple Genetic Algorithm” [47] is explained here. [Simple Genetic Algorithm \(SGA\)](#) was originally proposed by Holland [48]. The algorithm’s pseudo-code is shown in Algorithm 2.1. The details are presented here with minor modifications and additions to help with the explanation.

### 2.4.1 Initialization

The algorithm begins with the [initialization](#) operation, where a population of size  $P$  is randomly generated. The [population](#) is made up of [individuals](#). Considering a binary encoding, each individual is represented by a binary string or an array, composed of characters '0' and '1', e.g. an individual 010011. Using biology terminology, each character here is a [gene](#) and, the individual is referred to as a chromosome. Any other initialization related to the algorithm would happen here.

### 2.4.2 Encoding

[Encoding](#) is the mapping of the problem variables to the individual in the GA. As Srinivas & Patnaik [47] point out, the encoding is dependant on the variables of the optimization problem, where variables could be continuous, binary, etc. Here, we explain binary encoding.

The binary string represents the variables of the problem being solved. A simple example of such an encoding is a binary number.

### 2.4.3 Evaluation and the fitness function

In the [evaluation](#) operation, the current population is evaluated using a [fitness function](#). For an optimization problem, this could be the objective function or a combination of the objective function and the constraints.

### 2.4.4 Selection operation

[Selection](#) operation involves choosing individuals based on their fitness for reproduction. The higher the fitness value of an individual, the higher the probability that the individual will be chosen for reproduction.

### 2.4.5 Crossover operation

[Crossover](#) operation reproduces the selected individuals to create offsprings. Here, two parents are chosen for producing two children until a new generation of size  $P$  is generated. The process works by picking a random crossover point and then mixing parts of the parent strings to form the children. e.g. for parents: 11110 and 10001, with a randomly picked crossover point being after the first two characters, the resulting children would be 11001 and 10110.

The crossover process also relies on [probability of crossover \( \$p\_c\$ \)](#), which determines whether crossover will happen. If the crossover doesn't happen then the parents are copied directly as new population.

### 2.4.6 Mutation operation

[Mutation](#) operation follows the crossover operation. The [probability of mutation \( \$p\_m\$ \)](#) is a similar parameter as probability of crossover and determines whether mutation of a character in the individual's binary string happens [47]. Mutation changes the character from 0 to 1 and vice-versa. Here, one parent after mutation produces a child.

### 2.4.7 Termination criteria

The children produced from selection, crossover and mutation form the next generation. This next generation starts the next iteration following the aforementioned process.

Creation on new generations continue until the [termination criteria](#) has reached. An example of the termination criteria is [Iterations \(I\)](#) count, which is the number of generations of the initial population. The fittest individual from the last generation is chosen as the solution.

## 2.5 Conclusions

A short background of cloud computing, virtualization technology, performance evaluation and genetic algorithms has been presented in this chapter. Since, the thesis incorporates LQN performance models and genetic algorithms for solving provisioning problems, these have been discussed here. This chapter also explains “layered bottlenecks” [\[16\]](#), which is a bottleneck detection approach that has been incorporated in a provisioning algorithm that we have developed [\[15\]](#) (Chapter [4](#)).

## Chapter 3

# Cloud Provisioning for Meeting Performance

## Goals: Taxonomy and Survey

This chapter provides a detailed literature survey on dynamic provisioning within cloud systems. The discussion begins with listing the well-known types of provisioning and clarifying their terminology. Following this, those important research works that focus on this domain through scaling of applications with performance goals are mentioned. A very detailed classification follows, which views provisioning from different perspectives, aiding in understanding the process inside-out. This chapter explains provisioning by considering resources, stakeholders, techniques, technologies, algorithms, problems, goals and more.

**Chapter Outline** The structure of this chapter is as follows: Section [3.1](#) gives a general introduction to cloud provisioning, providing a detailed and general cloud provisioning classification, which views provisioning from different perspectives, aiding in understanding the process inside-out. Following this, the article explains cloud provisioning by considering decision-making entities, problems, types, scaling, schemes, policies, resources, techniques, technologies, algorithms and more. Section [3.2](#) mentions about the role of cloud provider and application providers in making decisions associated to provisioning. Section [3.3](#) mentions different provisioning types, scaling, associated problems and the placement schemes. Sections [3.4](#), [3.5](#), and [3.7](#) explain dynamic provisioning based on reactive and proactive provisioning, service-levels, and algorithms. Section [3.8](#) lists works that relate to dynamic provisioning and its progression to provisioning in the clouds with focus on performance. Section [3.9](#) presents our observations. Section [3.10](#) presents the conclusions.



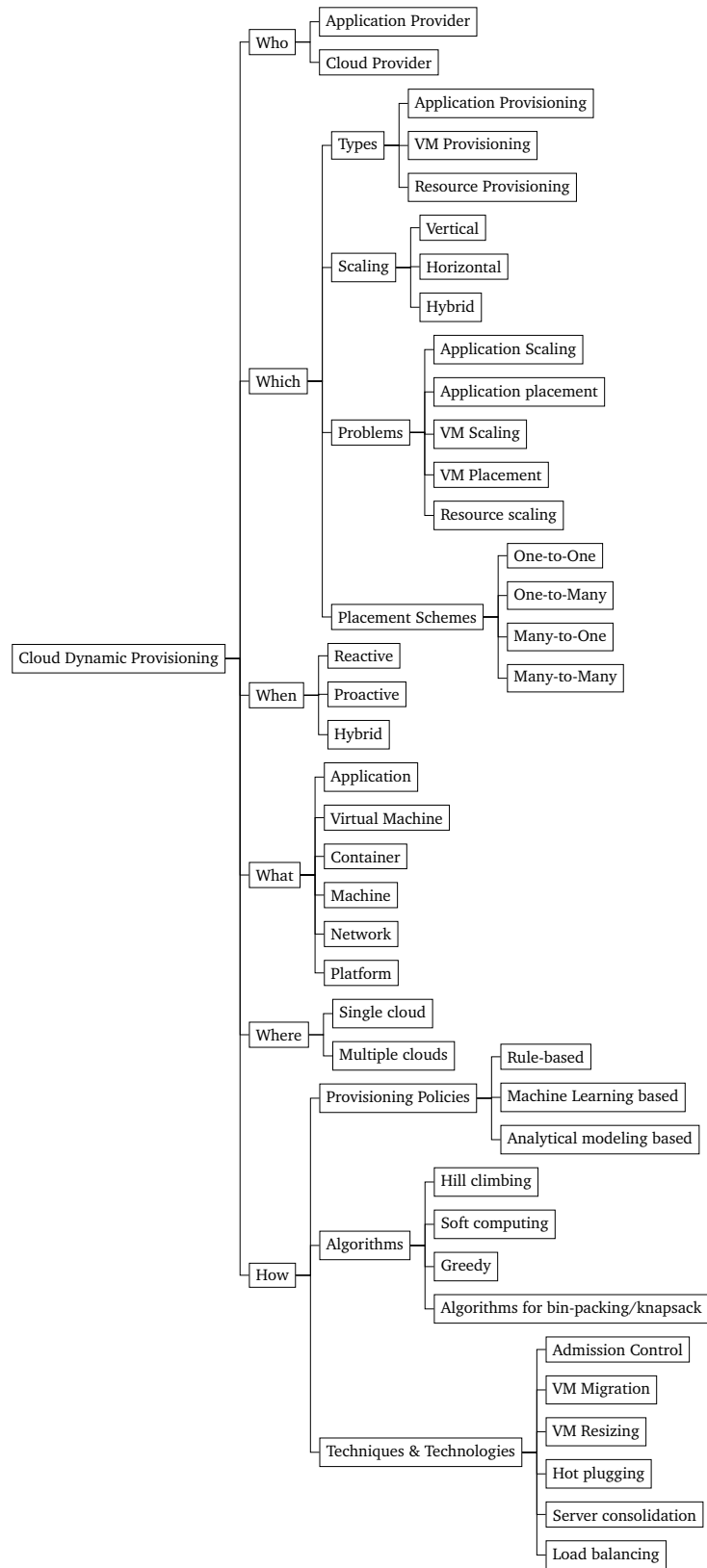


Figure 3.1: Cloud Provisioning - Classification

### 3.1 Classifying cloud provisioning

*Cloud provisioning* is a dynamic provisioning problem applied within a cloud system. Calheiros et al. [50] list following as three *steps* — or *types*, as referred in this article — to cloud provisioning:

1. **VM provisioning:** creation of VMs to meet software and hardware based requirements of an application such that given performance levels are achieved.
2. **Resource provisioning:** association of created VMs to adequate hardware resources.
3. **Application provisioning:** application deployment in the VMs and the subsequent association of the requests received to those applications.

There are various ways to classify provisioning in cloud systems. Following is the proposed classification (refer Figure 3.1 [51]):

- **Who** performs the provisioning (*the decision making entity*). These would be entities that make decisions about when resources need to be added and removed. AP and CP are two such entities. Section 3.2 explains the different decision making entities.
- **Which** provisioning *types, scaling, problems and schemes* exist. In a simple classification, provisioning may be classified as application, VM and resource provisioning. Related to these provisioning types are problems such as application scaling, application placement, VM scaling, VM placement and resource scaling. Section 3.3.1 and Section 3.3.2 explain these in detail. Furthermore, resource scaling may be classified either as horizontal or vertical, where the former type of scaling is based on adding or removing of new or existing resources (e.g. VMs) and the latter is associated with adding of resources to the existing live VMs and machines. Section 3.3.3 discusses about horizontal and vertical scaling. The relationships between resources and placement schemes are discussed in Section 3.3.4. One-to-one, one-to-many, many-to-one, and many-to-many schemes could be used to define different placements of VMs on physical machines and of applications on VMs.
- **When** is the provisioning decision made (*reactive versus proactive provisioning*). This could be after the workload and demands have increased/decreased or before the occurrence of such events. Reactive provisioning suggests provisioning decisions that are made after changes in workload behaviour is noticed whereas proactive provisioning involves prediction based approaches that help prepare ahead of changes in the workload and system usage. However,

a hybrid approach may also be applied for building a more robust system, using both reactive and proactive provisioning, as Section 3.4 explains.

- **What** *resource* is being provisioned and allocated, e.g. resources such as applications, VMs, processors, memory, network, storage, etc. Here, applications are allocated on VMs and VMs are allocated on physical machines. Also, there is inter-dependence between resources, e.g. in some cases increasing of processors may be achieved only by instantiation of a new VM and increase in VMs would increase all the resources that the new VM requires for execution. Sections 3.5 and 3.8 discuss the provisioning of resources as services.
- **Where** are the *resources deployed*. Location of resources is an important factor worth considering because of its effect on performance. A particular application may be deployed within a single cloud or on multiple clouds. Since a single cloud may have various data centers across the world, precise knowledge of placement of the application is required. In multiple clouds, having a core-and-edge-cloud architecture, deployment of the application presents new challenges, which is explained in Section 3.6.
- **How** is the provisioning problem being solved, relating to the various *policies, algorithms, techniques and technologies* that could be employed for deriving solutions. Commonly used algorithms include hill-climbing, soft computing techniques (e.g. genetic algorithms, neural networks, fuzzy logic etc.), greedy algorithms, bin-packing based algorithms, and knapsack based algorithms, etc. The main techniques and technologies that are applied and form a part of the algorithms are hot-plugging, VM migration, VM resizing, performance modeling, workload prediction, admission control, system monitoring, server consolidation and load balancing, which is , which is explained in Section 3.7.

The detailed classification is useful in understanding of cloud dynamic provisioning. In the following sections, we first discuss the types, problems and placement schemes associated with provisioning, and then mention works in cloud provisioning.

## 3.2 Who does cloud provisioning?

The stakeholders of cloud dynamic provisioning are: CPs, APs and end-users. These stakeholders are better described as roles and are defined with reference to the AP. APs, are the customers of the

CPs and provide software for use by the end-users. They have access to the services provided by the CPs but are unaware of the underlying cloud platform or cannot directly access the underlying hardware and platform. The role of the AP is also context-specific. If a CP consumes the services of another CP and then provides a software service to the end-user, then in this particular context, the former assumes the role of the AP. In this thesis, the discussion involves those APs who have control over the provisioning decisions of their application tasks onto VMs and container instances.

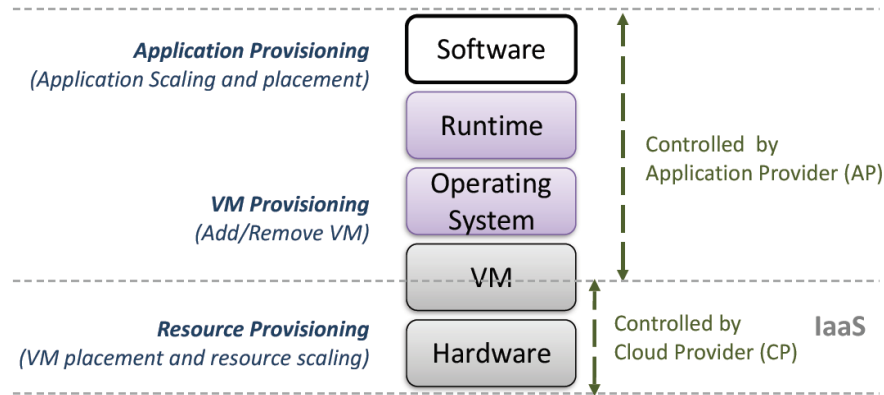


Figure 3.2: Cloud control hierarchy in an IaaS offering, displaying layers controlled directly by the CP and AP. CP chooses the hardware to allocate VMs on (resource provisioning). AP controls the application placement (application provisioning), addition/removal of VM instances (VM provisioning), and thread management. Although, CP houses the resources and as such has control over all layers, it does not directly administer services purchased by the customers. In the paper by Calheiros et al. [50], VM and application provisioning are performed by the AP and the authors mention that resource provisioning decisions are made by IaaS provider (i.e. CP). This image is presented here with modifications to the image in [15] © 2012 IEEE

Figure 3.2 depicts an **IaaS** offering, where the CP provides infrastructure to AP. Examples of such offerings are [Amazon EC2](#) [52], [Google Compute Engine \(GCE\)](#) [53], and [Microsoft Azure IaaS](#) [54]. The infrastructure includes hardware and virtualized resources such as VMs that run on the hardware. CP also make various VM-types available to the AP. These VM-types are heterogeneous VMs with different configurations of computing power and memory. In this context, AP are IaaS customers, who can create and instantiate VMs and install their choice of operating environment, which includes operating systems (OS) and software development kits (SDK). These VMs may be allocated and deallocated on-demand through facilities, such as command-line tools and API, provided by the CP. AP develop their applications and deploy them on the VMs, thereby providing software to their customers: end-users. Provisioning decisions made by the AP are for accommodating the workload demands of applications that are accessed by end-users.

Instead of a **IaaS** offering, we may have a **CaaS** offering. Figure 3.3 depicts a general **CaaS** offering, where the CP provides a containerized environment for the AP. Examples of such offerings

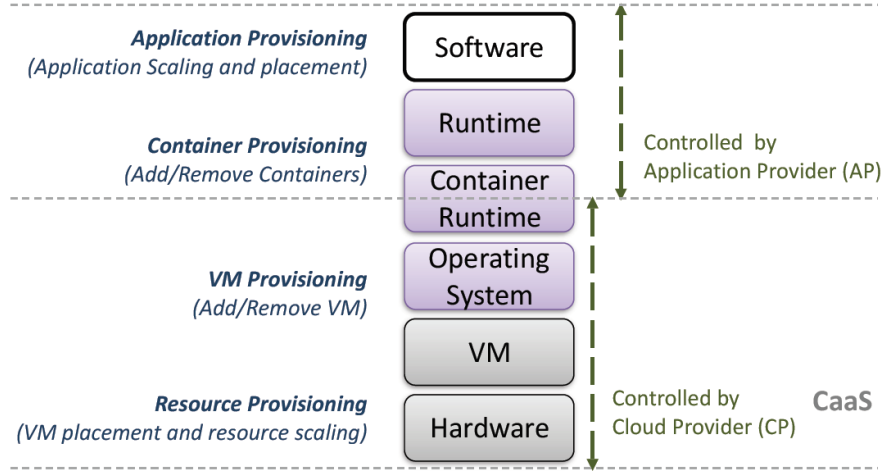


Figure 3.3: Cloud control hierarchy in a CaaS offering, displaying layers controlled directly by the CP and AP. CP controls provisioning of the underlying infrastructure for the containers to run on. AP controls the provisioning of containers. Each application service is separately deployed on a container, working together to represent the complete application. This image is presented here with modifications to the image in [15] © 2012 IEEE

are [Google Kubernetes Engine \(GKE\)](#) [55], [Azure Kubernetes Engine \(AKS\)](#) [56], [Amazon Elastic Container Service \(ECS\)](#) [57], and [Amazon Elastic Container Service for Kubernetes \(EKS\)](#) [57]. **CaaS** offerings are dependant on the underlying container orchestration mechanism used by the CP. In some cases the unit of scaling is a container whereas in other cases the unit is a higher level of abstraction that includes containers. In Kubernetes [58], the unit of scaling is a pod, which is a group of containers. [Amazon EKS](#), [AKS](#) and [GKE](#) are examples of Kubernetes based **CaaS** offerings. In **CaaS** offerings, APs decide on the number and types of containers and/or number of pods on which each of their services run.

The dynamic provisioning solution presented in this thesis can be applied for both offerings; in an IaaS offering the decision making will be with respect to VMs, whereas in a CaaS offering the decisions will be made for containers. For simplicity, throughout the thesis we will use the term VM provisioning for both type of compute instances: containers or VMs.

Cloud provisioning is the responsibility of [AP](#) and [CP](#). CP provision infrastructure. Provisioning decisions made by the AP are for accommodating the workload demands of applications that are accessed by end-users. Furthermore, CP can also deploy their own software and provide SaaS to end-users, in which case they take on the role of AP, in this context.

Each *application* is composed of running *tasks* (or processes). One or more tasks may be grouped together to form a *tier* of the application. A request that the application receives is handled by the

first tier and propagated to the next lower tier if further processing is needed, thereby having the request traverse through to the multiple tiers of the application, from one tier to the next. Once the request has been processed by the tiers, the response is sent back to the upper tiers and finally a response is returned to the end-user that initiated the request. In some works, a *service* may also be used to denote an application or its component, e.g. task.

### 3.3 Which provisioning types, scaling, problems and placement schemes exist?

In this thesis, we differentiate between the decision making process (i.e. solving the provisioning problem) and the action of provisioning. Furthermore, in the context of cloud provisioning “problems”, this thesis distinguishes between VM provisioning problem(s) and resource provisioning problem(s). Therefore, the problems associated with VM provisioning will decide the increase or decrease in the number of VMs and the problems associated with resource provisioning will determine the subsequent placement of the to-be-instantiated VMs on physical machines and the scaling of physical resources. The placement of VMs on physical machines is also known as “VM placement” [59,60] or “VM packing” [59]. Once, solutions to these problems have been generated, the decision maker informs the provisioner of the required actions that need to be taken and then the actual provisioning occurs.

In addition to above discussion, there is another reason for distinguishing between the two problems. When an AP decides to provision resources and makes an API call to instantiate a VM, it is the CP who decides on which particular physical machine would the instance reside, thus separating the decision making of the number of VM instances from their placement. Calheiros et al. [50] and Quiroz et al. [61] also make the distinction between VM provisioning and resource provisioning.

The following subsections discuss about provisioning problems and their association with application, VM and resource provisioning.

#### 3.3.1 Provisioning Problems

In this article, we consider dynamic provisioning to be associated with five *problems*:

1. **Application scaling (*AppScale*)**: determining the increase and decrease in the number of applications or units that applications are composed of (e.g. by addition of replicas of processes or application tiers).
2. **Application placement (*AppPlace*)**: determining the placement of application units on VMs or on physical machines; the latter is for when VMs are not employed.
3. **VM Scaling (*VmScale*)**: determining the increase and decrease in the number of VMs and associated resources such as virtual CPUs, memory etc.
4. **VM placement (*VmPlace*)**: determining the placement of VMs on the machines, i.e. the allocation of resources to the VMs.
5. **Resource Scaling (*ResScale*)**: determining the increase and decrease in the number of operating resources, which the applications and/or the VMs will run on and utilize.

### 3.3.2 Provisioning types

The different provisioning types are associated with the five problems identified above. Table 3.1 describes the relationship. As mentioned earlier, application and VM provisioning are the responsibility of AP, and CP is responsible for resource provisioning. If CP deploys the application then they adopt the role of the AP and therefore would control application and VM provisioning too.

Table 3.1: Dynamic provisioning and associated Problems

Provisioning	Associated Problems	Responsibility
Application Provisioning	<i>AppScale</i> AND/OR <i>AppPlace</i>	AP
VM Provisioning	<i>VmScale</i>	AP
Resource Provisioning	<i>VmPlace</i> AND/OR <i>ResScale</i>	CP

Figure 3.4 describes in detail cloud provisioning from a problem solving perspective. The figure shows  $n$  applications deployed on the cloud. Each application is composed of  $t$  tasks that have a dependency structure between each other representing the [application architecture](#). The applications are run on  $v$  VMs, which further run on  $p$  physical machines. The mapping of the

applications onto VMs solves the application provisioning problem. The tasks of the applications may have replicas and would be placed on the VMs. The determination of the number of VMs and its associated resources solves the VM provisioning problem. Finally, the resource provisioning solution places the VMs on various physical machines. Application receives requests from client, which is the workload, and as this workload fluctuates over time, the dynamic provisioning system manages and modifies the quantity and placement of resources accordingly. In this dynamic provisioning problem, solutions to the aforementioned problems would be used throughout the execution of the applications to determine any adjustments required in the system when receiving varying inputs.

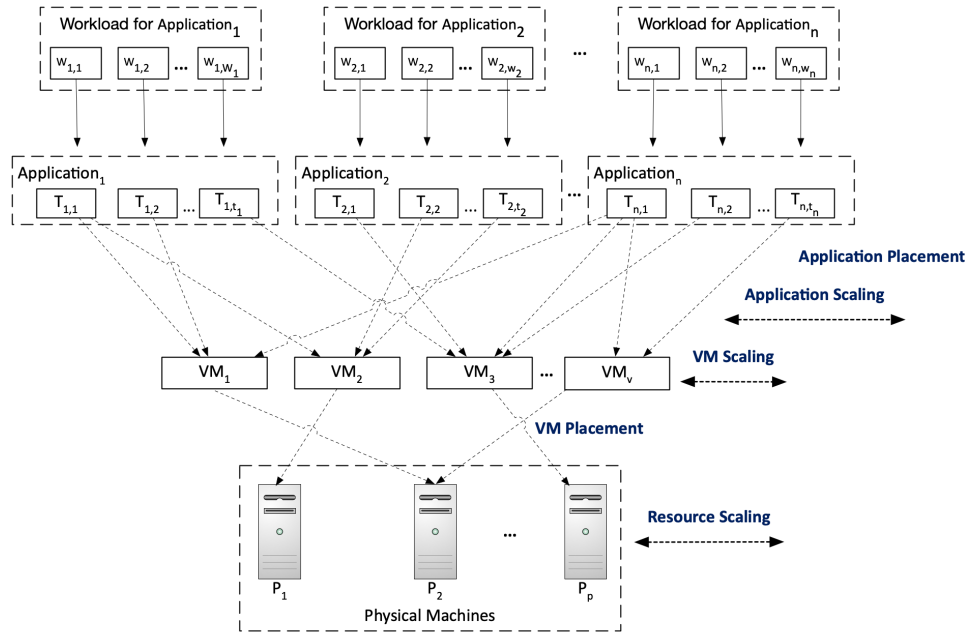


Figure 3.4: This figure shows applications managed by an AP within a cloud. Also refer to “Dynamic resource allocation” image that shows a similar idea of provisioning by Nguyen Van et al. [59], although more details are present in the above picture. AP are responsible only for Application placement, Application scaling and VM Scaling. CP are responsible for VM Placement and Resource Provisioning

It is key to realize that both VM provisioning and VM placement are “NP-hard knapsack” [59] problems, as characterized by Nguyen Van et al. [59]. These knapsack problems also relate to bin packing. The application placement problem has similarities to the VM placement problem.



### 3.3.3 Horizontal and Vertical scaling

[Autoscaling](#) is performed by either horizontal scaling or vertical scaling mechanisms. In horizontal scaling, a system is scaled through changes in the number of servers, e.g. replication of VM instances and addition of load balancers [62]. Likewise, vertical scaling denotes the changes to resources that are associated to an existing and running server, e.g. addition of processors to a live VM instance [62]. Vertical scaling is made possible through “hot plug” [63], a feature which allows for “dynamic” [64] changes to the devices (e.g. CPU) connected to a running system, without a system shutdown. The Linux kernel supports the hot plug feature [63, 65]; however, Vaquero et al. [62] point out that changes owing to vertical scaling occur on live servers and majority of “common operating systems” [62] do not support manipulating the CPUs and memory available to the system without a system reboot.

In spite of the lack of support from various operating systems, as noted above, research has begun to focus on hot plug support (e.g. [64, 66]) and on vertical scaling (e.g. [67]). Notably, VMware vSphere virtualization platform [68] added a hot plug feature in version 4.0 known as “hot add” [69] for adding processors, memory and virtual disks to a VM instance, which is also present for version 5 of the platform [70].

One advantage of vertical scaling over horizontal scaling as pointed by Yazdanov and Fetzer [67] is performance, where “VM instance acquisition time” [67] is longer in horizontal scaling. As research advances in hot plug technology, more performance improvements would also be realized in vertical scaling. However, there are limits to how much a system can scale vertically, in which scenario horizontal scaling could be adopted, as done by Yazdanov and Fetzer [67] in their design.

### 3.3.4 Placement schemes

The placement of applications on VMs and the placement of VMs on machines are governed by their placement schemes, which are described by the relationships that exist between the resources or entities. In this thesis, we introduce general relationships and notations that help in expressing these placement schemes. It is important to note that defining relationships and having a placement policy affects how the provisioning algorithm works and how the resources are scaled, although, defining a placement policy does not necessarily mean a placement algorithm is proposed.

Table 3.2: Placement schemes

Notation	Relationship	Description
$A \xrightarrow{1..1} B$	one-to-one	each A may be placed on one-and-only-one B
$C \xrightarrow{1..\infty} D$	one-to-many	each C may be placed on multiple D's, but multiple C's shall not reside within each D
$E \xrightarrow{\infty..1} F$	many-to-one	multiple E's may reside on one F, but these E's shall not span across multiple F's
$G \xrightarrow{\infty..\infty} H$	many-to-many	multiple G's may reside on one H, and each G may span multiple H's

Table 3.2 lists the relationships and their notations. In these relations, the entity on the left-side is placed onto the entity on the right-side. To express application placements, the entities on the left may be substituted by either application, tier or task, whereas in context of VM placement, the left-side entities would be VM. Following this, the right-side entity would be VM for application placement and host machine for VM placement. In the table,  $A \xrightarrow{1..1} B$ , describes a direct *one-to-one* relation between A and B, and that each A may be placed on one-and-only-one B, e.g. each application is allocated it's own VM or in case of VM placement, each VM would run on a separate host machine.  $C \xrightarrow{1..\infty} D$ , describes a *one-to-many* relation between C and D, where each C may be placed on multiple D's, but multiple C's shall not reside within each D, e.g. a dedicated hosting scenario where the components of an application — such as tasks or tiers — are placed across many VMs, but the VMs are not shared among different applications, and each VM would thereby include components of one application only.  $E \xrightarrow{\infty..1} F$ , describes a *many-to-one* relation between E and F, where multiple E's may reside on one F, but these E's shall not span across multiple F's.  $G \xrightarrow{\infty..\infty} H$ , describes a *many-to-many* relation between G and H, where multiple G's may reside on one H, and each G may span multiple H's, e.g. a shared hosting scenario where each application is spread across multiple VMs, and each VM is hosting multiple different applications. These relationships can be used to describe complex schemes and these notations can be extended further to accommodate more specific relationships or restrictions that may exist.

Calheiros et al. [50] in their VM and application provisioning approach have considered a “one-to-one mapping relationship between an application instance ... and a VM instance” [50]. Van et al. [59] in their work mention that an application can be associated to many VMs, and each VM is related to one application, which is similar to “dedicated hosting” [12]. The relation

between applications and VMs is important one to determine when dealing with application placement problems. Our work considers many-to-many placement scheme between tasks and VMs for [SatisfyQoS](#), and considers one-to-many mapping scheme between application tasks and VMs for [PGA](#).

### 3.4 When is the provisioning decision made?

Proactive provisioning is made possible through use of workload predictors, performance models and system monitors (e.g. [\[50\]](#)). Through workload prediction [\[71\]](#), the future incoming workload can be predicted and this would feed as input to the performance model. The parameters of the performance model would be determined through monitoring the system and through prediction modules [\[50\]](#). The performance metrics thus obtained from solving the model would indicate if constraints would be satisfied through the current system configuration. If the objectives are not met then the system configuration is tweaked through the performance model, following an iterative process until the objectives are satisfied or there is an indication that the objectives cannot be met. The system configuration that led to meeting of the objectives by solving the performance model is then applied to the real system such that the system can handle the incoming workload. In contrast to proactive provisioning, reactive provisioning proceeds with modification of system configuration as the workload changes. Example of reactive provisioning includes work by Iqbal et al. [\[60\]](#), who discuss a provisioning prototype implementation in the Eucalyptus Cloud, which horizontally scales the system based on the bottleneck tier found. Amazon auto scaling [\[72\]](#) provided by Amazon web services also allow for reactive addition/removal of VM instances in case of increase/decrease seen in the CPU resource utilizations. More recently, through machine learning, Amazon auto scaling allows proactive provisioning.

Urgaonkar et al. [\[12\]](#) propose both reactive and proactive/predictive provisioning approach for their provisioning algorithm. Although, their work is not applied to cloud systems, their approach is still applicable to the cloud provisioning domain. With proactive technique the dynamic changes to the system configuration are based on workload prediction from previously seen long-term workload demands. The reactive technique complements the previous approach by adjusting the configuration when sudden burst in workload is seen occurring within a short time duration.

Zhang et al. [4] have also emphasized the importance and necessity of both reactive and proactive methods.

### 3.5 What resources are provisioned?

Various resources are provided by clouds as services:

1. VMs and associated resources are provisioned through [IaaS](#)
2. Containers and associated resources are provisioned through [CaaS](#)
3. Development platform is made available through [PaaS](#)
4. Software is accessible through [SaaS](#)

In Section 1.2 various scaling policies were discussed. For the benefit of the APs, the CPs make various autoscaling policies available by default such that the APs can easily enable them for their applications. For VM autoscaling, such policies include rule based policies, schedule based policies (e.g. [6,7,73]), and machine learning based policies (e.g. [11]). Autoscaling is also made available for containers (e.g. [74,75]).

The CPs also allow APs to manually scale VMs and containers through the use of interfaces such as command-line tools and APIs (e.g. [76,77,78]). This allows for APs to use their own custom provisioning policies and cater to their specific requirements.

In this thesis, we propose custom dynamic provisioning solutions, which would be applied to the cloud through the use of the interfaces exposed by the CPs. The decisions made by the [decision maker](#) feed into the [provisioning agent](#) for actioning the changes. For this thesis, except the decision maker, the other components in the dynamic provisioning control system are out-of-scope. Section 5.2.2 explains the dynamic provisioning system. [provisioning agent](#) maps the [deployment configuration](#) suggested by the [decision maker](#) into the actual [deployment configuration](#) on the cloud.

The dynamic provisioning problems and solutions in the thesis can be applied for both VMs and containers. For Kubernetes, we assume a “one-container-per-Pod” [79,80] model, which is the general use case in Kubernetes. Furthermore, although there is a difference in how quickly VMs and containers can be provisioned, the decision making solution proposed in this thesis is not affected by this because we depend on the [decision scheduler](#) to decide when to invoke the [decision maker](#)

for decision making.

In the remainder of this section, we discuss provisioning of platform and other resources. Vaquero et al. [62] discuss about scaling of both servers and platforms in their paper with focus toward applications. They mention that rather than scaling based on each VM only, the other approach would be a controller that manages resources at a higher level of abstraction, more specifically by considering a complete application. With regards to scaling of platforms, the authors discuss about containers and databases. For scaling containers, component state has to be taken into consideration, and approaches may include state-aware components, replication of state information or caching. For databases, the following approaches may be adopted: “distributed caching, NoSQL databases, and database clustering” [62].

Alongside, other resources such as physical servers [81, 82] and network [62, 83] may be also be provisioned depending on them being made available as services by the CP and on state-of-art of research in these fields. “Metal as a service” (MAAS) [81] — which allows for provisioning of physical server nodes directly instead of VMs — is already offered by Canonical Inc. either by a package installation on Ubuntu operating system or through a newly installed Ubuntu Server. With regards to network, Vaquero et al. [62] discuss about the importance of scaling networks, pointing that in “consolidated datacenter scenario” [62] because many VMs use a shared network, a lot of network bandwidth may eventually be required. The usual approach adopted in this scenario is toward resource over-provisioning to meet (network) demands, which the paper argues against, as the method shows a lack of consideration toward applications that do not always consume the complete network bandwidth (allotted), hinting that such an arrangement leaves the network resources under-utilized. The authors suggest toward solutions that take “actual network usage” [62] into consideration, either by measuring network used by each application or by having the applications themselves ask for more network bandwidth. As cloud computing use grows, other resources would eventually come to be delivered as cloud services.

### 3.6 Where are the resources deployed?

Within a single cloud, an application is deployed on VMs, containers or software development platforms, however, there are other considerations such as distribution of the components across

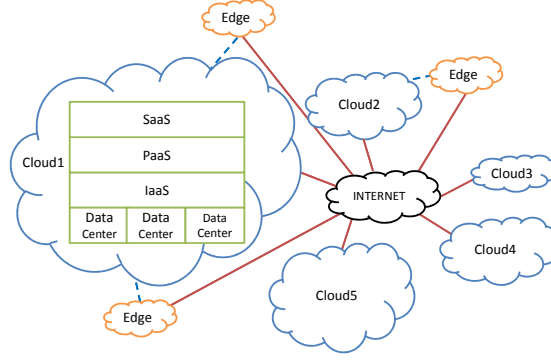


Figure 3.5: An application may be deployed on multiple clouds. Each “core cloud” [90] also has connected “edge clouds” [90] that help serve end-users for achieving better performance. For brevity edge clouds are only shown connected to Cloud1 and Cloud2 in the figure above.

different data centers and regions that need to be decided. The decision making here is dependant on what the CP chooses to deliver through its cloud platform. The platform may offer the AP to provide such details optionally or may completely leave such details for the CP to finalize. For example, Amazon EC2 [84] uses regions and availability zones for placement of resources. At present, nine regions are provided, which are spread across four continents. Multiple availability zones are situated in a region and the zones within a region are connected to each other. In case of Microsoft Azure [85], the platform is available in 17 regions and 141 countries.

Recently, the focus of the research community is moving from deploying resources on a single cloud to multiple clouds (a.k.a. “[m]ulti-clouds” [86]). Figure 3.5 depicts such as architecture, where resources are hosted across multiple clouds [86, 87]. There exists core clouds — similar to the clouds we know about today — that are connected to edge clouds, the latter being smaller in size than the core and located closer to end-users for achieving better performance [86]. Furthermore, efforts have also been made that consider development and deployment of services in such multi-clouds. e.g. Benfenatki et al. [88] include service provider and location (region and country) as preferences when deploying/developing a cloud service. Ferry et al. describe why “development and administration” [89] of applications across multiple clouds is a challenge. The main problem originates from the heterogeneity of the cloud solutions, which cause incompatibility and vendor lock-in, along with development challenges. To solve this they propose a model-based framework, where a generic deployment model can be used.

## 3.7 How are the provisioning problems solved?

In the following subsections we mention the different provisioning policies, algorithms, techniques and technologies that facilitate provisioning in cloud systems.

### 3.7.1 Provisioning policies

In Section 1.2, an introduction to provisioning policies was presented. In this section, we elaborate on those policies.

In general, CP provide AP the ability to specify rule-based policies on a system metric or a custom derived metric. Example of system metrics are CPU utilization, memory utilization, number of threads, queue length, etc., whereas, custom derived metrics are those metrics generated by the AP [7]. A simple example would be a set of the following two rules for dynamically scaling up and down compute instances as per their load:

1. Add two compute instance — step-size is two — when average CPU utilization in the instance group exceeds 80%.
2. Remove two compute instances when average CPU utilization in the instance group falls below 20%.

Well-known CP autoscaling policies such as Amazon EC2 Schedule Scaling [6], Microsoft Azure Autoscale [7], and Google Cloud Platform Autoscaling [73], enable APs to specify such rule-based policies. Rule-based policies may also be time-based (a.k.a. schedule-based) [7], e.g. add fifty compute instances at 9 a.m. and tear-down those fifty compute instances by 5 p.m.

Qu et al. [8] in their survey, point to the following shortcomings of simple rule based policies:

- these policies warrant the need to properly understand the application behaviour to correctly set the thresholds and corresponding actions
- these policies are incapable of adjusting to fluctuating application workload

Suggestions to improve these concerns have been provided in literature by using dynamic step-sizes and thresholds [8].

Machine learning (ML) based policies have also been used for solving dynamic provisioning problems (e.g. [9, 10, 11]). Using data gathered from the AP's use of the compute instances and

by using trained models, Amazon Predictive Scaling [11] forecasts and manages the provisioning of compute instances dynamically. Scaling options include optimization for availability (keeping CPU utilization at 40%), balance of cost and availability (keeping CPU utilization at 50%), and optimization for cost (keeping CPU utilization at 70%). The model can forecast for the next 48 hours.

However, ML based policies suffers from poor “initial performance” [9]. These models require time to converge and stabilize, causing the “auto-scaler to perform poorly during the active learning period” [8]. Also, the learning time is difficult to determine [8].

Analytical modeling overcomes the limitations posed by rule-based policies [9]. The analytical modeling approach also does not see the learning challenges faced by machine learning. In the domain of dynamic provisioning, analytical performance models are often built on queueing theory, where network of queues — Queueing Networks (QN) — become essential for modeling multi-tier applications to describe their [application architecture](#) [8].

With regards to provisioning and modeling of multi-tier applications, the easier approach is to split the overall metric (e.g. response time), to the metric of each service or each tier, and solve the problem (e.g. [12, 13, 14]), but this “results in suboptimal solutions globally” [8]. The more challenging approach is to consider the metric for the whole application and find the optimal deployment configuration “holistically” [8].

Although, powerful in their use for dynamic provisioning, analytical models exhibit lower accuracy in comparison to other evaluation methods. However, these are quicker to create and solve, and alongside are easier to manage [37]. With reference to analytical models, it has been pointed out that “accurate estimation of scaling models often require detail knowledge of service and workload characteristics thereby limiting their applicability to handle heterogeneous service mix” [9]. To address this concern, this thesis uses [LQN](#) analytical performance models [18]. LQN models have been used widely in many performance studies (e.g. [20, 21, 22, 23]). Shoaib and Das [22] have detailed the various studies on the use of LQN modeling for studying of software and web systems. LQNs are ideal for representing the interactions and intricacies of multi-tier applications and this work therefore uses LQN for performance modeling.

In this thesis, an analytical model based decision maker is built for dynamic provisioning of multi-tier applications. Here, we review the related work and explain the differences.



Zheng [91] presents a framework that automatically allocates application and database server resources when response time objectives are violated. Their controller uses hill-climbing algorithm. In comparison, our approach does not perform an incremental search but instead uses genetic algorithms with bottleneck analysis for decision making.

Urgaonkar et al. [12] propose a combination of workload prediction, virtualization technology, admission control, proactive and reactive provisioning to meet given response time deadlines of multi-tier systems. The QN model receives inputs from workload prediction to find “the number of servers to be allocated to each tier based on the estimated workload” [12]. Use of VMs help in quick adjustment of resources to the application tiers. In this particular work they consider a “dedicated hosting” platform [12], where each server runs one application and each application may spread across multiple servers. This work predates provisioning within cloud computing and does not consider cost, which is considered in our work, and as pointed by Qu et al. [8], breaks the response time per tier, rather than considering the response time of the whole application.

Van et al. [59] in their dynamic provisioning approach differentiate VM provisioning from VM placement and handle them in separate steps. Their approach deals with solving the cloud-based provisioning problems considering both performance and cost, where these problems as expressed as constraint satisfaction problems. Techniques for facilitating provisioning include instantiation and destruction of VMs, VM migration, and VM horizontal and vertical scaling; where vertical scaling is achieved through resizing the VMs. In comparison to our work, they employ an “empirical performance model based on experimental data” [59], whereas we use LQN models to drive the decision making.

Calheiros et al. [50], aim to satisfy QoS requirements such as response time, utilization and rejection rate of VMs based on negotiated QoS attributes. The solution employs workload prediction, performance modeling and VM monitoring. A simple QN model of the system is depicted. Furthermore, admission control is adopted by determining the maximum queue size of a queueing station and rejecting requests that arrive if the queue is full. The simulation results obtained by using CloudSim [92], for a 1000 host data center, show promising results where the VM hours were reduced — with no or about none rejection rate — and the negotiated response times were met. In contrast to their work, our work considers heterogeneous VMs with monetary cost and is aimed towards multi-tiered web applications, where each application may span multiple VMs.

Li et al. [93] have developed “CloudOpt” [93], an approach that relies on various methods such as network flow models (NFM), performance models and bin packing heuristics for solving problems associated with application, VM and resource provisioning in the clouds. The optimization problem is formulated as a mixed integer problem, which is derived from the NFM, and the performance models are used to include resource contentions. They consider a mapping between an application task and a VM; however, application and VM scaling together is achieved by creating replicas of tasks, whereas application and VM placement is achieved by moving those replicas between hosts. Their approach is applicable to dynamic provisioning environments when the CloudOpt optimization is “carried out periodically” [93]. The focus of their work is on meeting response time objectives and minimizing cost, while also considering software licence and memory constraints. Li et al. [94] improve on their previous work on “CloudOpt” [93], by restricting excessive changes caused by a new deployment configuration. Their work assumes a one-to-many mapping between task and VMs (as a result of task replication). Overall, their work differs from our work in the following two aspects: first, their work considers placement of VMs on hosts which is not considered by our work from an AP’s perspective and second, their work does not consider VM heterogeneity.

Shoaib and Das [15] (Chapter 4) present *SatisfyQoS*, a cloud provisioning algorithm that adds VMs or virtual processors based on the identified software and hardware bottlenecks, while considering various limits, such that specified performance constraints are met. The results from a case study show that using of simple bottleneck detection approach based on utilization of processors in comparison to the “layered bottlenecks” [16] approach leads to use of more resources to meet performance objectives. In comparison to *PGA* (Chapter 5.3 and Chapter 6), *SatisfyQoS* assumes a many-to-many mapping scheme between application task and VMs, whereas *PGA* assumes a one-to-many mapping scheme. Furthermore, cost is not considered in the work. For further differences between these two algorithms, please refer to Section 5.4.1.

Huber et al. [95] use Descartes Modeling Language (DML) based control loop for dynamic provisioning. The DML model is converted to LQN model for determining the steady-state results, and to SimQPN model for computing response times. In comparison to our work, their work does not consider cost of different VM types.

Han et al. [17] have considered dynamic provisioning using analytical performance modeling

for multi-tier applications with focus on performance and cost. In comparison to our work, their follow an incremental improvement of the configuration state, whereas we adopt a genetic algorithm based solution.

Overall, previous works have either excluded the minimization of monetary cost (e.g. [12, 15, 95]), or have not addressed the heterogeneity of compute instances (e.g. [15, 50, 93, 94]), or have simply resorted to incremental addition or deletion of resources to tiers by starting with a given deployment configuration until an acceptable performance has reached (e.g. [15, 17, 91]).

Unlike such works, we use an evolutionary optimization technique to systematically explore the solution space of deployment configurations that minimizes the resource cost incurred by the AP, meets the performance constraints, and considers variation in computing power of resources.

### 3.7.2 Algorithms

Various algorithms have been suggested and applied to solve cloud dynamic provisioning problems and these are mentioned below. Although many have been proposed before the advent of cloud computing they are still very applicable to solving provisioning problems in the cloud domain.

Hill climbing along with performance models have been used by Menasce et al. [96] for finding optimal system configuration parameters. In dynamic provisioning, hill climbing algorithm has been applied by Zheng [91] for dynamic provisioning of web and database servers, for admission control and thread count management. Buyya et al. [97], in relation to finding solutions to the differing QoS and optimization goals of cloud services, mention that for such cases we have a multi-dimensional optimization problem. To solve such problems, “one can explore multiple heterogeneous optimization algorithms, such as dynamic programming, hill climbing, parallel swarm optimization, and multi-objective genetic algorithm” [97]. Greedy algorithms, genetic algorithm, and various vector packing algorithms have been applied by Stillwell et al. [98] to solve the resource allocation problem in virtualized platforms. Simulated annealing algorithm has been proposed by Pandit et al. [99] for solving resource allocation in the clouds. Furthermore, Pandit et al. [99] has modeled the problem as a variant of multi-dimensional bin packing problem. The relationship between [Application Placement Problem \(APP\)](#) with bin-packing and multiple knapsack problems has explained by Urgaonkar et al. [100]. They propose various approximation algorithms and heuristics such as First-Fit, Max-First, Best-Fit, Worst-Fit, etc. to solve the APP.

As part of provisioning, for load prediction various algorithms have also been proposed. Xiao et al. [101] have presented their own algorithm: “Fast Up and Slow Down (FUSD) algorithm” [101], that predicts the expected resource utilization and helps toward making more stable provisioning judgements. Alongside, they have also looked at linear autoregression models for prediction and provide a comparison with their algorithm. Other techniques for load prediction include using artificial neural networks (e.g. Chabaa et al. [102], Prevost et al. [103]).

### 3.7.3 Techniques and Technologies

Provisioning in clouds is made possible through support of various techniques and technologies such as hot-plugging, VM migration, VM resizing, performance modeling, workload prediction, admission control, system monitoring, server consolidation and load balancing. As discussed earlier in section 3.3.3, the hot-plugging technology makes vertical scaling possible. There are also interesting mechanisms such as: VM migration, VM sizing and server consolidation [104] that play a key role in provisioning and used mainly for CP-based resource provisioning. Xiao et al. [101] have relied on VM migration and server consolidation technologies in their dynamic provisioning approach. Calcavecchia et al. [105] have used VM migrations for load balancing between the hosts machines. Performance modeling to help make dynamic provisioning decisions have been employed by many including Li et al. [106], Huber et al. [107], Shoaib and Das [15] (Chapter 4), and Li et al. [93]. Calheiros et al. [50] in their solution have employed multiple techniques such as workload prediction, performance modeling and VM monitoring. Multiple techniques such as admission control, VM provisioning, multiple job queues, request priority and performance monitoring have been employed by Das et al. [108] to meet response times of requests sent to a cloud.

## 3.8 Toward performance-oriented cloud provisioning

Performance of computer hardware and software systems has been the focus of many works such as that those by Lazowska et al. [42], Smith and Williams [34], and Menasce et al. [109, 110]. These works have explained performance and described the performance modeling process. They have also emphasized the importance of considering performance within the system development

process such that performance objectives are satisfied when the system is ready. Such performance objectives may be “response time, throughput, or constraints on resource usage” [34]. Research in performance of web systems has been conducted by many researchers including Dilley et al. [111], Menasce et al. [109], Liu et al. [20], Ufimtsev and Murphy [21] and Urgaonkar et al. [40]. Menasce [96] in their dynamic system reconfiguration approach present a QoS controller design that uses hill-climbing algorithm, relying on monitored data and QN models to derive an optimal system configuration (e.g. thread count and maximum queue size) for meeting the objectives of response time, throughput and rejection probability of a multi-tiered electronic commerce website.

Gradually, the focus of research moved toward dynamic provisioning, which previous works have dedicatedly looked into, and these efforts have been extended to find their application in the cloud computing domain as well. The important works below highlight these efforts and show the gradual progression toward performance-oriented cloud provisioning.

Karve et al. [112] describe the design of a middleware platform and controller, where the number of application instances are dynamically adjusted and placed on machines as per the demands of the application. This work adds to their previous work [113], by achieving better balance of application load on the machines and providing improvements to their algorithm by minimizing number of placement changes — through the use of “incremental placement” [112] approach — and maximizing the achieved application demand. In their work they consider CPU and memory resource capacities of the machines and the respective requirements of the applications in making placement decisions.

Urgaonkar et al. [100] provide the algorithms that solve the APP for applications running on clusters and show that these problems are NP-hard. They describe how APP relate to bin-packing and multiple knapsack problems. Online and Offline APP as two types of placement problems that are discussed. Offline APP compute placements without taking the order of the applications into consideration, whereas online APP compute placements one-by-one for each application, placing applications from lower to higher indices, and not allowing changes in placement of applications that were placed earlier than the application that is currently being considered for placement. Depending on the variants of online and offline APP, as per the different placement restrictions, they propose various approximation algorithms and heuristics such as First-Fit, Max-First, Best-Fit, Worst-Fit, etc. to solve the APP.

Urgaonkar et al. [12] propose a combination of workload prediction, virtualization technology, admission control, proactive and reactive provisioning to meet given response time deadlines of multi-tier systems. They laid a strong foundation for future research work on provisioning in the clouds and is one of most detailed and significant contributions before research in cloud provisioning began in earnest.

Iqbal et al. [60] discuss their application and VM provisioning prototype implementation, which they have extended from their earlier work [114] on one-tier to two-tier web applications, in the Eucalyptus Cloud. First, the bottleneck tier is found using a simple algorithm based on CPU utilizations and response times, and then VMs are added such that response times guarantees are not violated. They further plan to improve their algorithm to include n-tier applications.

Kijsipongse et al. [115] propose an architecture which dynamically provisions VMs from a remote cloud according to one of the two provisioning policies introduced by the authors and adds the VM as part of a local cluster (management) system. A test-bed is setup where Eucalyptus cloud is used, which can host a maximum of six VMs on the cloud. The focus of this work is on the job scheduling and VM provisioning policies with related affect on job queue size.

Li et al. [106] use Layered Queueing Network (LQN) performance models [116], which are analytical models based on extended QN, along with network flow model (NFM) to solve the deployment optimization problem in maximizing the profit of the cloud while considering response time or throughput as constraints. Their combined application, VM and resource provisioning approach meets objectives by optimally placing VMs on physical machines, where each application task is hosted on one VM.

Apart from previous works, Calheiros et al. [50] present a unique perspective where VM and application provisioning is done by those providing both SaaS and PaaS (i.e. AP), whereas resource provisioning is reserved for IaaS providers, mainly due to the lack of “control” [50] regarding VM placement available to the AP. Similar idea of “decoupled control” [117] has previously been suggested by Lim et al.

In their proposed algorithm, Calheiros et al. [50], aim to satisfy QoS requirements such as response time, utilization and rejection rate of VMs based on negotiated QoS attributes. The solution employs workload prediction, performance modeling and VM monitoring. A simple QN model of the system is depicted. Furthermore, admission control is adopted by determining the

maximum queue size of a queueing station and rejecting requests that arrive if the queue is full. The simulation results obtained by using CloudSim [92], for a 1000 host data center, show promising results where the VM hours were reduced — with no or about none rejection rate — and the negotiated response times were met.

Chi et al. [118] provide a heuristic search algorithm to find optimal deployment configuration for a multi-tiered web application in the clouds. A given configuration includes the number of VMs allocated to each tier. The main features of the algorithm include a utility function that adds another level of QoS along with response time, a rule-set database to find initial configuration for a given workload, and a pruning algorithm which finds the optimal deployment. The pruning algorithm adds/removes VMs of tiers of an application and uses differences in workloads and the utility function to reach optimal solution quicker. Alongside, performance models are used for decision making. They evaluate their approach with a two-tier web application. For finding optimal configuration they use an expert system — relying on and updating the rule-set database — and their pruning algorithm.

Li et al. [93] have developed “CloudOpt” [93], an approach that relies on various methods such as network flow models (NFM), performance models and bin packing heuristics for solving problems associated with application, VM and resource provisioning in the clouds. The optimization problem is formulated as a mixed integer problem, which is derived from the NFM, and the performance models are used to include resource contentions. They consider a mapping between an application task and a VM; however, application and VM scaling together is achieved by creating replicas of tasks, whereas application and VM placement is achieved by moving those replicas between hosts. Their approach is applicable to dynamic provisioning environments when the CloudOpt optimization is “carried out periodically” [93]. The focus of their work is on meeting response time objectives and minimizing cost, while also considering software licence and memory constraints.

Shoaib and Das [15] (Chapter 4) present a detailed provisioning algorithm that adds VMs or virtual processors based on software and hardware bottlenecks identified, while considering various limits, such that the performance goals specified and met. The results from a case study show that using of simple bottleneck detection approach based on utilization of processors in comparison to the “layered bottlenecks” [16] approach leads to use of more resources to meet performance

objectives.

Das et al. [108] meet response times of requests sent to a cloud through employment of admission control, VM provisioning, multiple job queues, request priority and performance monitoring. They use simulation performed using CloudSim [92] to show their provisioning approach performs better than a simple VM provisioning approach, by showing results on admitted/served requests, rejected requests and VM instantiation time.

Calcavecchia et al. [105] present “Backward Speculative Placement (BSP)” [105], a technique that processes VM deployment requests received by a CP to optimize the placements of VMs on heterogeneous physical machines. The objectives considered are meeting of the CPU demands of the VMs, minimizing the VM migrations and balancing of the load between the hosts. The technique uses two phases for the decisions, where one phase handles new requests for VM deployment and another phase periodically optimizes the existing placement (through VM migrations). Through simulation results they show that their approach generates placements that meet “high level of demand satisfaction” [105]. Owing to VM migrations, the number of physical machines that are active changes in their problem, and therefore we consider their approach to also includes solving of the resource scaling problem along with the VM placement problem.

Casalicchio et al. [119] also focus on the VM placement problem faced by CP. The objective is maximize the revenue of the CP while meeting resource, availability and VM migration constraints. Their approach presents an algorithm based on hill climbing to solve the problem.

Xiao et al. [101] in their article explain their approach to dynamic VM placement and resource scaling in the clouds. The aim is toward placing VMs on machine such that the requirements of VMs are met and also toward minimizing the number of machines used. Through VM migrations the overload on machines is decreased and idle machines are turned off (or set to standby). Their approach is based on a load prediction algorithm (“Fast Up and Slow Down (FUSD) algorithm” [101]) and a “skewness” [101] metric; the former is used for the prediction of expected resource utilization, and the latter is related to the utilization of multiple resources on each machine and is minimized in the algorithm. They present results from both simulation and measurements to demonstrate the applicability of their approach.

Pandit et al. [99] have proposed a simulated annealing algorithm for resource allocation in the clouds and model this problem as a variant of multi-dimensional bin packing problem. They explain



their bin packing problem modeling through a simple VM placement example. Through simulation they show that their proposed algorithm performs better than First-Come-First-Serve algorithm by having a higher average resource utilization when mapping the requests to the resources. We consider that their approach relates to resource allocation problems in general, such as application placement and VM placement.

Stillwell et al. [98] have proposed algorithms for resource allocation in virtualized shared platforms running homogeneous machines. Although their approach focuses on static workloads, by periodically finding the allocations, their approach would be applicable for dynamic workloads in dynamic provisioning scenarios. They formulate the problem of resource allocation as a mixed integer linear program (MILP), where the objective is to maximize performance and fairness through a metric known as “minimum yield” [98]. The article is mostly focused on allocation of services to machines, where each service runs within one VM, although services that run on multiple VMs are also included in the discussion. The proposed algorithms include greedy algorithms, genetic algorithm, and vector packing algorithms (multi-dimensional bin packing). Since the article focuses on resource allocation problem in general, it is applicable to solving both VM placement and application placement problems.

Alongside with VM provisioning, researchers have also considered how applications relate to VMs. Calheiros et al. [50] in their VM and application provisioning approach as discussed above, have considered a “one-to-one mapping relationship between an application instance ... and a VM instance” [50]. Van et al. [59] in their work mention that an application can be associated to many VMs, and each VM is related to one application, which is similar to “dedicated hosting” [12]. The relation between applications and VMs is important one to determine when dealing with application placement problems.

Including performance, dynamic provisioning could be employed to achieve various goals. Researchers have used their approach to meet multiple objectives such as minimizing cost and meeting performance objectives (e.g. [59, 67, 106]), or addressing availability and performance (e.g. [120]), or considering energy and performance (e.g. [121]).

Chapter 5 also presents a VM provisioning solution using new genetic operators to meet performance constraints and minimize cost. Readers may refer to Section 1.2 for further details on the novelty of the solution.

### 3.9 Observations

This survey article explains cloud provisioning and its different types. Cloud provisioning is used for various purposes, including but not limited to following: minimizing resource usage, minimizing cost, meeting QoS, achieving multiple objectives, where algorithms should be quick, cause minimal changes to the existing allocation scheme, and handle different resource types.

A detailed classification of cloud dynamic provisioning is also presented, followed by mentioning of key related research work that contributed to dynamic provisioning approaches in general and recent research efforts in application, VM and resource provisioning. Along with the classification, this thesis contributes by explaining the different facets of provisioning in the clouds and in particular explaining in detail the following: reactive and proactive provisioning, horizontal and vertical provisioning, provisioning of resources as services, provisioning goals, policies, algorithms, techniques and technologies involved.

The following observations are made through the survey:

- Many publications do not explicitly or clearly mention the provisioning and placement schemes that are employed in their work. This is necessary for clarity. This survey introduces notations that can serve a useful purpose for explaining and identifying the schemes used by a provisioning scheme.
- Performance is key in the adoption of cloud computing where every factor such as network delay, VM size, application and cloud architecture play an important role in deciding how the deployed applications on the cloud perform. However, what events trigger the provisioning decision-making entity and how frequently should provisioning happen to maintain stability of the system, has not been given much attention. Research should be conducted to help AP easily make such decisions.
- Interest in multi-clouds is growing with intention of reaping benefits from such an architecture. Alongside, this promotes the viewpoint of developing and testing software where services are deployed in unknown locations but are still connected together to perform various tasks. However, not enough research exists that describe the challenges and solutions of adopting multi-clouds. Future research should investigate the on-going challenges of developing software for multi-clouds and compile a list of best practices to be followed in

this field.

- To track performance of applications and the system, and to alert providers in case of exceeding thresholds or occurrence of incidents, integration with effective and efficient profiling, monitoring and notification services has become essential for cloud platforms. A monitoring and notification system should be in place to monitor if performance goals are being met, and research should help identify the best architecture and practices to follow for establishing such a system.

### **3.10 Conclusions**

This chapter begins with explaining cloud provisioning and its different types, alongside clarifying the terminology used to describe them. This is followed by mentioning key related research work that contributed to dynamic provisioning approaches in general and recent research efforts in VM and application provisioning. A detailed classification of dynamic cloud provisioning is presented. The chapter explains different facets of provisioning in clouds and particularly explains in detail the following: reactive and proactive provisioning, horizontal and vertical provisioning, provisioning of resources as services, provisioning goals, policies, algorithms, techniques and technologies involved.

## Chapter 4

# VM provisioning using layered bottlenecks

This chapter presents results from the initial work done on VM provisioning using layered bottlenecks. Most parts of this chapter have been published in [15], whereas some parts have been modified and added.

Here, we present a VM provisioning problem for an AP that requires meeting the performance constraints, given an initial [deployment configuration](#). As a solution, we present a controller algorithm based on performance models that adds VMs or virtual processors depending on the software and hardware bottlenecks identified. The results from a case study show that our approach that relies on the use of “layered bottlenecks” [16] in comparison to a simple bottleneck detection approach based on just the utilization of processors ends in the use of fewer resources.

**Chapter Outline** Following is an outline of this chapter. Section 4.1 establishes the assumptions made for the research and formulates the problem. Section 4.2 presents the solution to the problem, by explaining the controller algorithm ([SatisfyQoS](#)) and a provisioning framework using the controller. Section 4.3 describes the case study and results. Section 4.4 presents the conclusions.

### 4.1 The Provisioning Problem

Figure 4.1 describes the different entities that constitute a cloud application provisioning problem. These entities are: an application managed by an AP, the tasks of the application and the VMs on which the tasks run. The problem’s goal is to meet the mean response time constraints in response to the changing workloads. The response time constraints are set by the AP as per the requirements of the application.

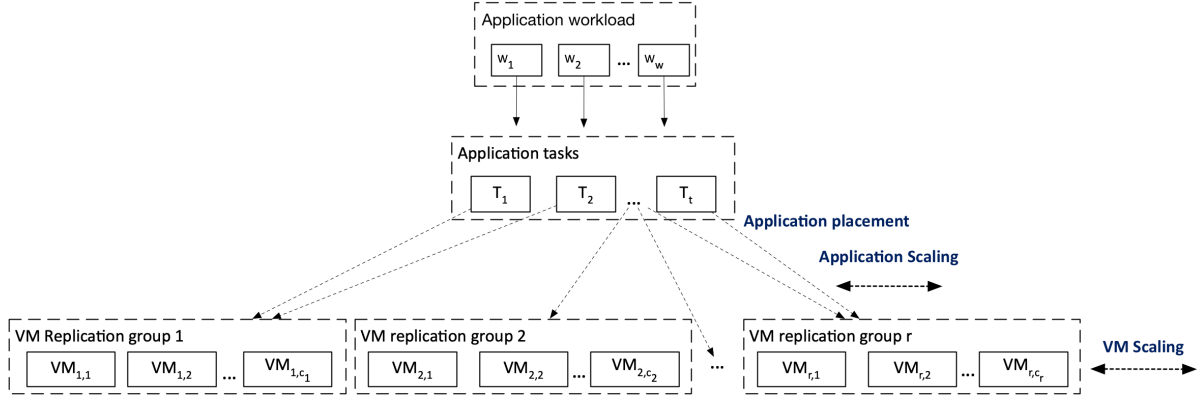


Figure 4.1: This figure shows an application that is managed by an AP within a cloud. Based on the problem that is being solved, there is a many-to-many relationship between application tasks and the VMs, and the VMs can be replicated.

The following subsections formulate a formal problem definition, beginning with the assumptions, then notations and finally presenting the formulated problem.

#### 4.1.1 Assumptions

Following are assumptions we make regarding VMs, containers, applications, and tasks.

##### 4.1.1.1 VM replication - horizontal scaling

A VM runs on a virtual processor of one or more multiplicity, i.e. a multi-processor, and one or more application tasks may run on a VM. There is a many-to-many relationship between application tasks and the VMs. A VM could be replicated, distributing the incoming workload between the replicated VMs.

##### 4.1.1.2 Vertical scaling

Instead of replicating a VM, the processors can also be added to the running VMs. This allows for vertical scaling when the processor is the bottleneck.

##### 4.1.1.3 Current deployment configuration

The algorithm presented also requires a current deployment configuration. This is because the two provisioning decisions made by the algorithm — i.e. replicating VMs and adding virtual processors — need an existing configuration to begin with.

#### 4.1.1.4 Containers as compute instances

In this chapter we have referred to VMs as the compute instances, however, containers could also be used instead. This would be representative of a container based offering such as [GKE](#) and [AKS](#).

#### 4.1.1.5 Resource removal

To meet the performance objectives the presented algorithm adds resources. For a complete dynamic provisioning solution the removal of resources is also required when system is under utilized. To achieve this completeness, the algorithm can be combined with a simple utilization based algorithm. In this situation when resources are under utilized the utilization algorithm should be triggered and resources could be removed until response time constraints are met.

#### 4.1.2 Notations

Following explains the notations used in Figure 4.1:

$t$	number of tasks	(4.1)
$T_i$	task $i$ ,	
	where $1 \leq i \leq t$	(4.2)
$T = \{T_i \mid 1 \leq i \leq t\}$	set of Tasks	(4.3)
$w$	number of workloads classes	(4.4)
$W_i$	workload $i$ ,	
	where $1 \leq i \leq w$	(4.5)
$W = \{W_i \mid 1 \leq i \leq w\}$	set of Workload classes	(4.6)
$maxProcsPerVM$	maximum processor per VM	(4.7)
$maxVmReplicas$	maximum VM replicas per group	(4.8)
$maxVMs$	maximum VMs	(4.9)
$r$	number of VM replication groups	(4.10)
$c_i$ , where $1 \leq i \leq r$	number of VM replicas in group $i$	(4.11)

$$vp_{i,j} \quad \text{where } 1 \leq i \leq r, 1 \leq j \leq \text{maxProcsPerVM}$$

number of  $j$  processors in group  $i$  (4.12)

$$a_{i,j}, \quad \text{where } a_{i,j} \in \{0, 1\}$$

is task  $T_i$  allocated to group  $j$  (4.13)

$$A = t \times r \text{ matrix, where element at index } i, j \text{ is } a_{i,j}$$

matrix of task to replication group allocations  
represents a deployment configuration (4.14)

$$RT_{W_i} \quad \text{model mean response time for } W_i \quad (4.15)$$

$$RTobj_{W_i} \quad \text{mean response time constraint } W_i \quad (4.16)$$

$$sat_{thresh} \quad \text{saturation threshold}$$

for identifying bottlenecks (4.17)

### 4.1.3 Formal problem definition

Following the notations specified in Section 4.1.2, the problem is formally defined as:

$$\text{given } CurrentDeploymentConfiguration \ \& \ sat_{thresh} \quad (4.18)$$

$$\text{meet constraints } \forall W_l \in W : RT_{W_l} \leq RTobj_{W_l} \quad (4.19)$$

$$\forall i, k : vp_{i,k} = 0 \quad (4.20)$$

$$\forall i : 1 \leq c_i \leq \text{maxVmReplicas} \quad (4.21)$$

$$1 \leq \sum_{m=1}^r c_m \leq \text{MaxVMs} \quad (4.22)$$

$$1 \leq l \leq w \quad (4.23)$$

$$1 \leq i \leq r \quad (4.24)$$

$$k > \text{maxProcsPerVM} \quad (4.25)$$

The problem that we solve here is a “constraint satisfaction problem” [45], and it is stated as follows:

1. given the current deployment configuration and the  $sat_{thresh}$ .

2. for all workloads of the application the model's mean response time is less than the respective response time constraint.
3. number of VMs replicas should not exceed  $maxVmReplicas$
4. number of VM processors should not exceed  $maxProcsPerVM$

The decision variables of the problem are:  $r$ ,  $c_i$ ,  $vp_{i,j}$  and  $A$ .

The task-to-VM-type allocations are expressed as the matrix  $A$ , where each state of the matrix represents a deployment configuration.

## 4.2 The Solution

Here, a solution that uses “layered bottlenecks” [16] for VM provisioning is presented.

### 4.2.1 Layered bottlenecks solution

Provisioning resources by identifying bottlenecks through resource utilizations is one VM provisioning solution. In this case, bottlenecks would be resources that exceed a set threshold utilization. Once the bottleneck is found, instances of VMs or associated virtual processors are added to improve performance. However, applications running on the VM could also be bottlenecks instead of the virtual processor. In such scenarios it is essential to distinguish if the virtual CPUs or the software is limiting the performance and as an effect simplifying to whether virtual CPUs or the VM instances should be increased.

We use the notion of “layered bottlenecks” [16] that rely on the “BStrength” [16] metric to identify bottlenecks associated with software processes. Readers may refer to Section 2.3.3, which explains layered bottlenecks. A controller algorithm that bases the provisioning decisions on virtual processor utilizations and on BStrength of processes is presented in this chapter.

The controller uses LQN analytical models for finding bottlenecks and for decision making. The implementation of the controller algorithm is in Java and uses the [jLQNInterface](#) library. The following subsection introduces the library.



### 4.2.2 jLQNInterface Java library

The controller algorithm requires manipulating LQN models. For this purpose, from the LQN toolset LQX tool may be worth considering. LQX program allows for solving LQN model with varied values for one or multiple parameters. Although LQX is a useful tool, it lacks facilities to manipulate the structure of LQN models [122]. As Mroz and Franks [123] point out, there are advantages of having a well-defined static model structure, which mainly eases “accelerated convergence” [122], thereby decreasing the time to solve models iteratively. However, for our purposes we require manipulating LQN model structures, therefore, we have developed [jLQNInterface](#) [15], our own tool using Java programming language that allows solving, analyzing, and manipulating the LQNX models through its API. If models are in plain LQN model format then they are easily converted to LQNX format. Furthermore, the tool includes powerful features of object-oriented programming due to Java, is type-safe, portable and has easy to use API. The purpose of jLQNInterface is to serve as a building block for other tools and as a bridge — through its API — to allow for easy interaction with LQN models and the solver. The controller algorithm as described in Section 4.2 for solving the VM provisioning problem has been implemented using the jLQNInterface tool.

### 4.2.3 Contributions Overview

In this chapter, we focus on VM provisioning by the AP. The main contributions are VM provisioning using:

1. [jLQNInterface](#) library to solve, analyze and manipulate LQN models.
2. Controller algorithm based on “layered bottlenecks” [16] by:
  - (a) Adding VMs
  - (b) Increasing number of virtual processors
  - (c) Considering constraints such as:
    - i. Maximum VMs
    - ii. Maximum processors-per-VM
    - iii. Maximum replication-per-VM
  - (d) Having the current deployment configuration.
3. Case study demonstrating applicability of the algorithm

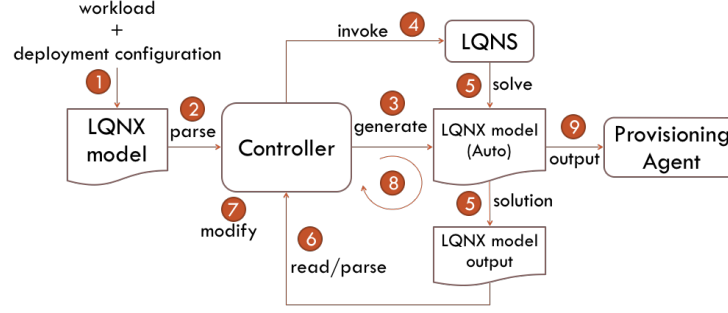


Figure 4.2: Single step process flow diagram of provisioning framework [15] © 2012 IEEE. This image is presented here with few modifications.

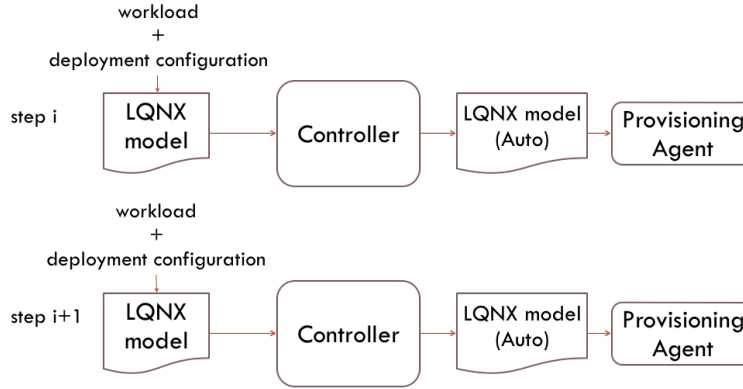


Figure 4.3: Inter-step process flow diagram of provisioning framework [15] © 2012 IEEE. This image is presented here with few modifications.

#### 4.2.4 Provisioning framework

In this section we describe the workings of a provisioning framework which relies on performance modeling and our controller algorithm to allocate resources through a provisioning agent (Figure 4.2 and Figure 4.3). The step begins with using the an XML-based LQN performance model (LQNX) provided by the AP, which is updated with the inputs: workload and [deployment configuration](#). The framework invokes the controller at regular intervals — known as “controller intervals” [96] — duration of which is decided through the [decision scheduler](#). The interval chosen should not be too close to cause frequent provisioning leading to instability and should not be too far apart to miss QoS targets. Each invocation of the controller algorithm by the framework at a given interval is considered as a *step* and each step comprises of multiple *loops*. Within each loop of a given step  $i$ , the following *actions* are processed (Figure 4.2 and Figure 4.3):

1. **Input:** Workload and [deployment configuration](#) from monitored data are served as inputs
2. **Read/Parse:** Model is read and parsed by the Controller, which saves the model information

in memory.

3. **Generate:** Controller generates a copy of the model from memory and stores it on disk. This *Auto model* is modified through the next set of loops in the step
4. **Invoke:** LQNS is invoked to begin model solving
5. **Solve:** LQNS solves model generating a LQNX output
6. **Parse Solution:** Result is then parsed by the Controller
7. **Modify:** Based on the model, result, response time objectives, and the provisioning algorithm, the controller modifies the model in-memory such that the current model state is closer to meeting QoS objectives.
8. **Repeat:** Actions 3–7 are repeated until the controller has either met QoS objectives or finds that objectives cannot be satisfied. If goals are met then action 9 is executed
9. **Provision:** From the model the [deployment configuration](#) is sent to provisioning agent.

#### 4.2.5 Controller Algorithm

This section presents the [SatisfyQoS](#) algorithm. *FindBottlenecks* and *FindMaxResourcePerf* are two methods called by *SatisfyQoS*.

The *SatisfyQoS* (Algorithm 4.1) begins by first initializing the internal data structures, which requires reading and parsing the input LQNX model. Along with the LQNX model, the saturation threshold and the response time constraint form the other inputs of the algorithm. The output is the model that meets the mean response time constraints.

After initialization, an approximation of maximum performance possible is found. This is accomplished by invoking the *FindMaxResourcePerf* algorithm. Franks et al. [16] outline a simple process of finding maximum resource performance; a similar approach is followed here. For all processors — except ones which represent client task processors and ones with infinite queueing stations (*InfProc*) — the multiplicity is set to the maximum of *maxProcPerVM* and *maxVmReplicas*. This corresponds to the maximum processor multiplicity possible. Similarly, for all tasks — except the tasks which represent clients (*RefTasks*) and the tasks that have infinite queueing stations (*InfTasks*) — the multiplicity is set to the product of *maxVmReplicas* and the current multiplicity of the task (thread count). After these changes, the model is stored from the memory of the controller to disk and solved. The response time found is returned to the *SatisfyQoS* algorithm.

**Input:** *InputModel*,  $sat_{thresh}$ , *rObjective*

**Output:** *OutputModel* that satisfies QoS

**Definitions:**

*addedVMs*: Current counter of VMs added

*rMin*: approximate minimum response time possible

```
1: INITIALIZE()
2:  $rMin \leftarrow \text{FINDMAXRESOURCEPERF}()$ 
3: if  $rMin > rObjective$  then
4:   return Unable to meet objectives
5: end if
6: INITIALIZE()
7: while true do
8:    $rTime \leftarrow \text{SOLVEPERFORMANCEMODEL}()$ 
9:   if  $rTime \leq rObjective$  then
10:    return Objective satisfied
11:   end if
12:    $bSet \leftarrow \text{FINDBOTTLENECKS}()$ 
13:   if  $bSet == \{\}$  then
14:    return Unable to meet objectives
15:   end if
16:   for all  $B$  in  $bSet$  do
17:     if  $B \in Tasks$  then
18:       if  $B.processor \in bSet$  then
19:         continue
20:       else
21:         if  $(B.VMreplica < maxVmReplicas$ 
22:            $\&\& addedVMs < maxVMs)$  then
23:           Add 1 VM replica of  $B.processor$ 
24:            $addedVMs++$ 
25:         else
26:            $immutable \leftarrow immutable \cup B$ 
27:         end if
28:       else if  $B \in Processors$  then
29:         if  $B.multi < maxProcPerVM$  then
30:           Increase  $B$  multiplicity by 1
31:         else
32:            $immutable \leftarrow immutable \cup B$ 
33:         end if
34:       end if
35:     end for
36:   end while
```

---

Once the minimum response time is received from the *FindMaxResourcePerf* algorithm, the *SatisfyQoS* verifies if the value is below the response time constraint. If not, then the algorithm

outputs that the model is unable to meet constraints and halts, otherwise the algorithm continues running.

Next, the main loop of the algorithm begins; however before this the internal data structures have to be reset back to the configuration of the actual model, which is achieved by again following the initialization process. Within the loop, the model is solved by invoking LQNS and the results are parsed to find the response time. If the response time is less than the objective, then the algorithm ends with a success. Otherwise, the bottlenecks are found by invoking the *FindBottlenecks* algorithm.

The *FindBottlenecks* (Algorithm 4.2) is a modified version of the original “layered bottlenecks” [16] algorithm (Section 2.3.3). The main difference is use of *immutable* set and the *isLayered* boolean flag by the *FindBottlenecks* algorithm. This algorithm receives as input the performance model,  $sat_{thresh}$ , *isLayered* boolean, and *immutable* set. Algorithm finds the layered bottleneck when *isLayered* flag is set to true. If *isLayered* boolean flag is true then *BStrength* is used for finding bottleneck from tasks after checking if a processor is the bottleneck, otherwise if false, bottlenecks are found based only on saturation values of resources. The *immutable* input contains all those resources whose multiplicity and replication cannot be increased because of reaching the limits of resources available and are therefore not considered when identifying bottlenecks. Also, processors with infinite queueing stations and tasks which are clients or have infinite queueing stations are not used in finding of bottleneck resources, as the goal here is to find bottlenecks in the cloud servers, not clients.

Any bottlenecks identified by *FindBottlenecks* is sent back to the *SatisfyQoS* algorithm. If the bottlenecks returned is an empty set, then the algorithm exits with an output that it is unable to meet constraints, otherwise the algorithm continues. For all the identified bottlenecks, it is checked if they belong to tasks or processors set. If the bottleneck is a task then the first check is to verify if the processor that the task runs on is not a bottleneck as well. If this is so, then this task bottleneck is disregarded and no action is performed on it; the reason being that any resource increments would be performed on the processor by increasing multiplicity in contrast to performing a VM replication. Otherwise, for the bottleneck task, if it’s processor is not the bottleneck, then replication of the task’s VM is performed. However, if the bottleneck is a processor then the multiplicity of the processor is incremented. Before performing VM replication or before adding processors, the limits:

---

**Algorithm 4.2** FINDBOTTLENECKS [15] © 2012 IEEE

---

**Input:** *InputModel*,  $sat_{thresh}$ , *isLayered*, *immutable***Output:** Bottleneck set: *bSet*

```
1: bSet = {}
2: for all  $p \in Processors$  where  $p \notin InfProc$  do
3:   Calculate  $sat_p$ 
4: end for
5: for all  $t \in Tasks$  where  $t \notin RefTask, InfTask$  do
6:   Calculate  $sat_t$ 
7:   Find  $Below_t$ 
8:   Calculate  $BStrength_t$ 
9: end for
10: if isLayered == true then ▷ Find layered bottlenecks
11:    $B \leftarrow \arg \max_{p \in Processors} sat_p, \text{ where } sat_p \geq sat_{thresh} \ \&\& \ B \notin immutable$ 
12:   if  $B \neq NULL$  then
13:     bSet = {B}
14:   else
15:      $B \leftarrow \arg \max_{t \in Tasks} BStrength_t, \text{ where } sat_t \geq sat_{thresh} \ \&\& \ B \notin immutable$ 
16:     if  $B \neq NULL$  then
17:       bSet = {B}
18:     end if
19:   end if
20: else ▷ Find bottlenecks based on saturation only
21:   for all  $p \in Processors$  do
22:     if  $sat_p \geq sat_{thresh}$  then
23:       if  $p \notin immutable$  then
24:         bSet  $\leftarrow bSet \cup p$ 
25:       end if
26:     end if
27:   end for
28:   for all  $t \in Tasks$  do
29:     if  $sat_t \geq sat_{thresh}$  then
30:       if  $t \notin immutable$  then
31:         bSet  $\leftarrow bSet \cup t$ 
32:       end if
33:     end if
34:   end for
35: end if
36: return bSet
```

---

*maxVMs*, *maxProcPerVM*, *maxVmReplicas* are checked, and if any limit is reached then instead of adding resources, the resource is noted as *immutable*. This information is later used when finding bottlenecks as detailed earlier.

Above describes one complete loop of the algorithm, and the next loop begins again with solv-

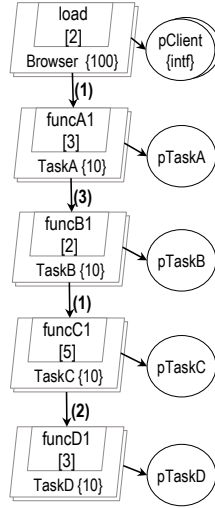


Figure 4.4: Case study input model. Minor modifications to [15] © 2012 IEEE

ing the performance model, finding bottlenecks and adding resources until either no bottlenecks remain — in which case the algorithm ends with output of unable to meet constraints — or the desired response time constraint is met — in which case the algorithm ends with output of meeting objectives. In the latter case, the final performance model is sent to the provisioning agent to allocate resources.

### 4.3 Case Study

A case study using the *SatisfyQoS* algorithm was conducted using a simple LQNX input model with aim to demonstrate the applicability of the proposed algorithm. One complete step of the algorithm, involving several loops, was run such that the mean response time constraint could be achieved. The following two cases were tried:

1. **Case 1:** *isLayered* is true, layered bottlenecks
2. **Case 2:** *isLayered* is false, bottlenecks are based on saturation values only

*isLayered* boolean has been discussed in Section 4.2.5. The input model and the result from the two cases have been presented below. LQNS version 5.4 was used for the evaluations.

Figure 4.4 shows the input model representing 100 Browser clients and four uni-processor VMs on the cloud, each running a single task. Each Browser request is sent first to TaskA. For a received request, multiple requests are sent further down to another task and this value is shown within

parentheses in the model, e.g. for each request TaskA task receives three requests are sent to TaskB. After processing a request the response is sent back to the task which initiated the request, e.g. TaskD would send responses to TaskC. The service demand of a request is shown within square brackets and the maximum thread count is shown within curly braces. The parameters were set as follows:  $sat_{thresh} = 0.8$ ,  $maxVMs = 20$ ,  $maxVmReplicas = 10$ ,  $maxProcsPerVM = 6$ . The input model response time was 1806.720 ms and desired response time was 350 ms.

Listing 4.1: Case 1 result based on Layered Bottlenecks [15] © 2012 IEEE. Image is presented here with further details

```

1 Met response Time Objectives - DONE
2 Objective: 350.000, ResponseTime: 302.942
3
4 Summary of changes:
5 Bottlenecks: [pTaskD]
6 Loop1 Change1: PROC: pTaskD multiplicity = 2 in next loop
7
8 Bottlenecks: [pTaskC]
9 Loop2 Change2: PROC: pTaskC multiplicity = 2 in next loop
10
11 Bottlenecks: [pTaskD]
12 Loop3 Change3: PROC: pTaskD multiplicity = 3 in next loop
13
14 Bottlenecks: [pTaskC]
15 Loop4 Change4: PROC: pTaskC multiplicity = 3 in next loop
16
17 Bottlenecks: [pTaskB]
18 Loop5 Change5: PROC: pTaskB multiplicity = 2 in next loop
19
20 Bottlenecks: [pTaskD]
21 Loop6 Change6: PROC: pTaskD multiplicity = 4 in next loop
22
23 Bottlenecks: [pTaskC]
24 Loop7 Change7: PROC: pTaskC multiplicity = 4 in next loop
25
26 Bottlenecks: [pTaskD]
27 Loop8 Change8: PROC: pTaskD multiplicity = 5 in next loop
28
29 Bottlenecks: [TaskB]
30 Loop9 Change9: TASK: TaskB VM = 2 in next loop
31
32 Bottlenecks: [TaskA]
33 Loop10 Change10: TASK: TaskA VM = 2 in next loop
34
35 Bottlenecks: [pTaskC]
36 Loop11 Change11: PROC: pTaskC multiplicity = 5 in next loop
37
38 Bottlenecks: [pTaskD]
39 Loop12 Change12: PROC: pTaskD multiplicity = 6 in next loop

```



```

40
41 Bottlenecks: [pTaskC]
42 Loop13 Change13: PROC: pTaskC multiplicity = 6 in next loop
43
44 Bottlenecks: [pTaskD]
45 Loop14 Limit reached: PROC: pTaskD not duplicated in next loop
46
47 Bottlenecks: [TaskC]
48 Loop15 Change14: TASK: TaskC VM = 2 in next loop

```

For Case 1, results from running the algorithm are shown in Listing 4.1. The algorithm executed 15 loops and performed 14 changes to meet the response time objective. The response time of the Browser task was 302.942 ms. The output model included 7 VMs and 24 processors.

For Case 2, results from running the algorithm are shown in Listing 4.2. The algorithm executed 5 loops and performed 19 changes in total to meet the response time objective. The response time of the Browser task was 299.537 ms. The output model included 14 VMs and 27 processors.

Listing 4.2: Case 2 result based on resource saturation [15] © 2012 IEEE. Image is presented here with further details

```

1 Met response Time Objectives - DONE
2 Objective: 350.000, ResponseTime: 299.537
3
4 Summary of changes:
5 Bottlenecks: [TaskA, TaskB, TaskC, pTaskC, pTaskD]
6 Loop1 Change1: TASK: TaskA VM = 2 in next loop
7 Loop1 Change2: TASK: TaskB VM = 2 in next loop
8 Loop1 Change3: PROC: pTaskC multiplicity = 2 in next loop
9 Loop1 Change4: PROC: pTaskD multiplicity = 2 in next loop
10
11 Bottlenecks: [TaskA, TaskB, TaskC, pTaskC, pTaskD]
12 Loop2 Change5: TASK: TaskA VM = 3 in next loop
13 Loop2 Change6: TASK: TaskB VM = 3 in next loop
14 Loop2 Change7: PROC: pTaskC multiplicity = 3 in next loop
15 Loop2 Change8: PROC: pTaskD multiplicity = 3 in next loop
16
17 Bottlenecks: [TaskA, TaskB, TaskC, pTaskC, pTaskD]
18 Loop3 Change9: TASK: TaskA VM = 4 in next loop
19 Loop3 Change10: TASK: TaskB VM = 4 in next loop
20 Loop3 Change11: PROC: pTaskC multiplicity = 4 in next loop
21 Loop3 Change12: PROC: pTaskD multiplicity = 4 in next loop
22
23 Bottlenecks: [TaskA, TaskB, TaskC, pTaskC, pTaskD]
24 Loop4 Change13: TASK: TaskA VM = 5 in next loop
25 Loop4 Change14: TASK: TaskB VM = 5 in next loop
26 Loop4 Change15: PROC: pTaskC multiplicity = 5 in next loop
27 Loop4 Change16: PROC: pTaskD multiplicity = 5 in next loop
28
29 Bottlenecks: [TaskA, TaskC, pTaskD]

```

```

30 Loop5 Change17: TASK: TaskA VM = 6 in next loop
31 Loop5 Change18: TASK: TaskC VM = 2 in next loop
32 Loop5 Change19: PROC: pTaskD multiplicity = 6 in next loop

```

Table 4.1: Summary: Case 1 and Case 2 results [15] © 2012 IEEE. The mean response time constraint is 350 ms.

	Input	SatisfyQoS - Case 1	Saturation - Case 2
VMs	4	7	14
Processors	4	24	27
Mean response time	1806.720 ms	302.942 ms	299.537 ms
Loops	-	15	5
Changes	-	14	19

Table 4.1 summarizes the results of Case 1 and Case 2. As seen, Case 2 result has a better mean response time than Case 1; however, the goal is to meet mean response time which is also satisfied by Case 1. Case 1 required more loops due to increase of resources one at a time rather than multiple resource increments as within each loop of Case 2. But Case 1 caused fewer changes and resulted in use of far less VMs and processors than Case 2. A possible reason is due to the search for bottlenecks after each change and selective addition in Case 1, where preference is given to only one resource with maximum saturation or **BStrength** value. Based on the results, Case 1 met the constraints with fewer resources.

## 4.4 Conclusions

In this chapter, a dynamic provisioning algorithm (**SatisfyQoS**) for adding resources is proposed that can be applied to cloud computing domain. The algorithm finds bottlenecks based on the notion of “layered bottlenecks” [16]. Depending on bottleneck type, either processors or VMs are added to help ease the bottleneck, thereby improving performance. Before incrementing, the limits such as maximum VMs, processors-per-VM, and replication-per-VM are checked and kept under bounds. This process of finding bottlenecks and adding resources is continued until the desired performance constraints are satisfied.

The provisioning algorithm has been implemented by using **jLQNInterface**, a multi-purpose tool developed by the author for solving, analyzing, and manipulating the LQN models. Furthermore, the **jLQNInterface** API serves as a building block and bridge for other utilities that intend on interfacing with LQN models and the solver.

A case study demonstrates the applicability of our proposed algorithm and use of layered bottlenecks in resource provisioning through a run of the implemented algorithm. Two cases were tried, where Case 1 used layered bottlenecks and Case 2 used saturation values to determine the bottlenecks. It is found that although Case 1 required more loops through the algorithm, the resulting model used fewer resources while meeting the desired constraints.

This work does not incorporate deallocation of VMs and processors. Furthermore, the algorithm also does not include cost in the decision making. In Chapter 5 a complete VM dynamic provisioning problem is presented to address these concerns.

# Chapter 5

## The Decision Maker

In the previous chapter, a [Constraint Satisfaction Problem \(CSP\)](#) problem for tackling VM provisioning to meet application performance constraints was introduced. In this chapter, the problem is updated and extended by allowing de-allocation of VMs, introducing VM-types, enforcing a specific task-to-VM allocation policy, removing vertical scaling of processors, and introducing a cost objective. Parts of this chapter have been accepted for publication in [124].

**Chapter Outline** This chapter is organized as follows. Section 5.1 details the dynamic provisioning problem that is being solved. Section 5.2 presents the solution overview and the choices of the four algorithms within the decision maker. Section 5.3 goes into the details of these algorithms. Section 5.4 is the discussion. Section 5.5 is the conclusions.

### 5.1 The provisioning problem

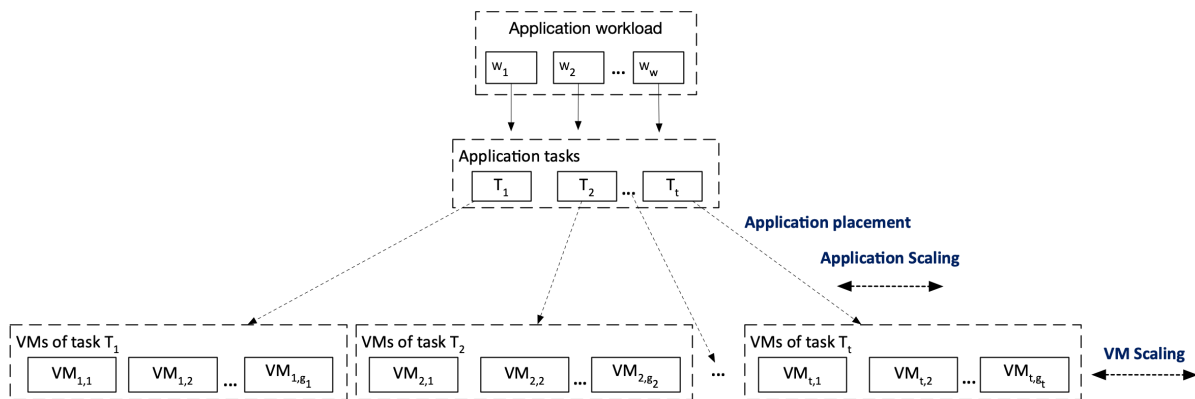


Figure 5.1: This figure shows an application that is managed by an AP within a cloud. The application is composed of tasks and these tasks are placed on VMs. There is a one-to-many relationship between tasks and VMs. As the workload that the application tasks receive, increase and decrease, the application and VMs also scale up and down, respectively. The details and decision of the scaling is made by the decision maker.

Figure 5.1 describes the different entities that constitute a cloud application provisioning problem. These entities are: an application managed by an AP, the tasks of the application and the VMs on which the tasks run. The goal of the problem is to minimize cost of the running application and meet the given mean response time constraint in response to the changing workload. The response time constraint is set by the AP as per the requirements of the application. If multiple applications are managed then the AP would have one constraint per application.

The following subsections formulate a formal problem definition, beginning with the assumptions, then notations and finally presenting the formulated problem.

### **5.1.1 Assumptions**

Following are the assumptions that have been held in formulating this problem.

#### **5.1.1.1 Cloud provisioning and the AP**

The problem assumes decision making is for an application running within the cloud or a similar virtualized environment that gives control to AP for instantiating different VM types. Here, AP manages the application and by minimizing the cost of the running application, the eventual goal is to minimize cost for the AP.

#### **5.1.1.2 Task-to-VM relationship**

In Figure 5.1 we see a one-to-many relationship between tasks and VMs, i.e. a task can be deployed on many VMs and that task is the only task that runs within that group of VMs. There are reasons for imposing this opinion. This follows the “one service per container” [125] pattern of microservices architecture, where services provide a small separate functionality and run within their deployment units. Similar pattern is being adopted when deploying software on the cloud as suggested by Sousa et al. [126]. Another benefit of adopting this pattern and thereby having a one-to-many relationship between tasks and VMs is that it reduces the complexity of the decision making, instead of considering a many-to-many relationship.

### 5.1.1.3 Containers as compute instances

In this chapter we have referred to VMs as the compute instances, however, containers could also be used instead. This would be representative of a container based offering such as [GKE](#) and [AKS](#).

### 5.1.1.4 Analytical performance model

To solve the dynamic provisioning problem, various [deployment configurations](#) need to be evaluated before finding the candidate that solves the provisioning problem. For evaluating the configurations LQN analytical performance models will be used.

Analytical models are not as accurate compared to real measurements but are quick in providing solutions [19]. This is specially useful for our purposes where we need to traverse and solve a large set of deployment configurations.

For correctness, the AP has to ensure that the model used has passed through the validation and verification phases of the modelling cycle [42].

### 5.1.1.5 Load balancing and VM replication

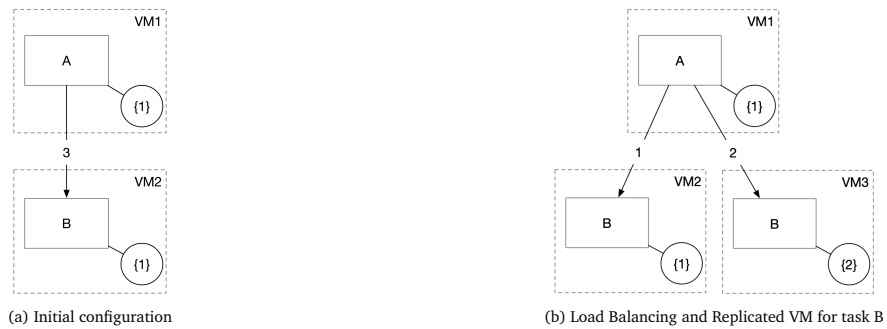


Figure 5.2: Load Balancing and VM replication

Consider two tasks A and B which run within their VMs as shown in Figure 5.2a. The tasks are represented here as rectangles with solid lines, and the VMs are represented as rectangles with dashed lines. A calls B three times for each request that it receives. Furthermore, A and B runs on a uni-processor shown with the circle attached to the tasks. Let's say to handle the increasing workload we decide to provision a new dual-processor VM for task B. Figure 5.2b shows the result of the replication. The load into B is split between VM2 and VM3 now, proportional to

the number of the processors associated with the VMs of task B. The opposite behaviour happens when de-provisioning VMs, where instead of splitting the load, the load is combined together.

For our purposes, initial configuration is not required. The implemented load balancing behaviour for the model is able to determine the distribution of the calls based on the configuration of the callee tasks. This load balancing behaviour is assumed when replicating task VMs. This same idea of VM replicas of a task is shown in Figure 5.1 and has been discussed above in Section 5.1.1.2.

#### 5.1.1.6 Non-replicable tasks

Some tasks cannot be replicated and the decision maker allows for such tasks to be excluded from scaling. Such non-replicable tasks are: “databases which do not support clustering or synchronization; [tasks] which are not designed to be distributed (singleton [tasks]); or [tasks] with expensive licenses” [127].

#### 5.1.1.7 Cost

The cost of the application is determined by the number of running instances each VM-type and the type’s cost rate (as a function of time). The VM-types and the cost rate can be changed depending on the CP from within the decision maker. For the provisioning problem, the goal is to minimize cost.

#### 5.1.1.8 Response time constraint and workload classes

To account for performance, the mean response time constraint is used. Figure 5.1 shows  $w$  workload class types and the output response time of each class is a constraint. Workload class types are same as “customer classes” [42].

To confirm that response time constraints are met, the response time results — for each workload class — obtained from solving LQN models are compared with the mean response time constraints set by the AP. These results and the constraints are expressed with the assumption that they are steady-state mean response times.

#### 5.1.1.9 Reference tasks

In LQN performance models, workload class types are represented as reference tasks. These reference tasks are not included within the set of application tasks and are excluded from scaling decisions. Furthermore, these reference tasks are assumed to run within the client environment and are hence not included in the cost.

#### 5.1.1.10 Maximum VMs per task

A constraint that considers the maximum allowed VMs per task, has also been added to the problem. The parameter: *MaxVmsPerTask* is to be set by the AP for this purpose. There are two reasons for introducing this constraint. First, this parameter has a direct relationship on the total possible configurations, and helps with restricting the options, and thereby speeding up the solution search. Second, this allows for a soft control over the deployed software cost.

#### 5.1.1.11 Provisioning type

The workload input in Figure 5.1 is assumed to change depending on the type of provisioning decision that is made. For [proactive provisioning](#), forecasted workload would be served as input to the decision maker. For [reactive provisioning](#) the current workload is used as input.

#### 5.1.1.12 Multiple applications

Figure 5.1 represents the dynamic provisioning problem for one application, indicating one decision maker per application. Each AP managed application has its own decision maker. Since the objective of each decision maker is to minimize cost for the application and meet the response time constraints, the end result of all decision makers is the minimization of cost for the AP while meeting response time constraints.

### 5.1.2 Notations

Following explains the notations used in Figure 5.1:

$$t \quad \text{number of tasks} \quad (5.1)$$



$$T_i \quad \text{task } i, \quad \text{where } 1 \leq i \leq t \quad (5.2)$$

$$T = \{T_i \mid 1 \leq i \leq t\} \quad \text{set of Tasks} \quad (5.3)$$

$$w \quad \text{number of workload class types} \quad (5.4)$$

$$W_i \quad \text{workload } i, \quad \text{where } 1 \leq i \leq w \quad (5.5)$$

$$W = \{W_i \mid 1 \leq i \leq w\} \quad \text{set of Workloads received by the application} \quad (5.6)$$

$$v \quad \text{number of VM-types available from the CP} \quad (5.7)$$

$$s_{i,j}, \quad \text{where } 0 \leq s_{i,j} \leq \text{MaxVmsPerTask} \quad \text{number of VM-type } j \text{ allocated to task } T_i \quad (5.8)$$

$$S = t \times v \text{ matrix, where element at index } i, j \text{ is } s_{i,j} \quad \text{matrix of task to VM-type allocations,} \quad \text{represents a deployment configuration} \quad (5.9)$$

$$g_i = \sum_{j=1}^v s_{i,j} \quad \text{number of VMs allocated to task } T_i \quad (5.10)$$

$$\text{MaxVmsPerTask} \quad \text{Maximum VMs that can be allocated to a task} \quad (5.11)$$

$$C_i \quad \text{cost for acquiring VM-type } i \quad (5.12)$$

$$\text{Cost} = \sum_{i=1}^t \sum_{j=1}^v s_{i,j} \times C_j \quad \text{cost of VMs} \quad (5.13)$$

$$RT_{W_i} \quad \text{Response time for } W_i \text{ (Performance model)} \quad (5.14)$$

$$RTobj_{W_i} \quad \text{response time constraint } W_i \quad (5.15)$$

$$RTobj = \{RTobj_{W_i} \mid 1 \leq i \leq w\} \quad \text{set of response time constraints} \quad (5.16)$$

$$(5.17)$$

### 5.1.3 Formal problem definition

Following the notations specified in Section 5.1.2, the problem is formally defined as:

$$\text{minimize } \sum_{m=1}^t \sum_{n=1}^v s_{m,n} \times C_n \quad (5.18)$$

$$\text{subject to } \forall W_l : RT_{W_l} \leq RT_{obj_{W_l}} \quad (5.19)$$

$$1 \leq g_i \leq MaxVmsPerTask \quad (5.20)$$

$$0 \leq s_{i,j} \leq MaxVmsPerTask \quad (5.21)$$

$$1 \leq i \leq t \quad (5.22)$$

$$1 \leq j \leq v \quad (5.23)$$

$$1 \leq l \leq w \quad (5.24)$$

The problem would be stated as follows: minimize the total cost of the application based task-to-VM-type allocations and VM-type costs, such that the following constraints are met:

1. for all workloads, the performance model's mean response time is less than the respective mean response time constraint.
2. number of VMs allocated to a task should not exceed *MaxVmsPerTask*
3. number of each VM-type allocated to a task should not exceed *MaxVmsPerTask*

The decision variables here are the task-to-VM-type allocations. These allocations are expressed as the matrix  $S$ , where each state of the matrix represents a deployment configuration.

The aforementioned problem of minimizing the application cost is a constrained non-linear non-convex integer optimization problem, for which heuristic algorithms are well suited to find near-optimal solution(s).

## 5.2 The solution

In Section 5.1, a cloud application provisioning problem was formulated. In this section, we present a solution to the problem using the decision maker, which is introduced in Section 5.2.1. Section 5.2.2 explains a dynamic provisioning control system that includes the decision maker component. Section 5.2.3 explains the internal workings on the decision maker. Section 5.2.4

mentions the encoding for the algorithms that are used by the decision maker. Section 5.2.5 shows the number of evaluations for the algorithms. Section 5.2.6 discusses about the algorithm parameters required before running the algorithms.

### 5.2.1 The decision maker

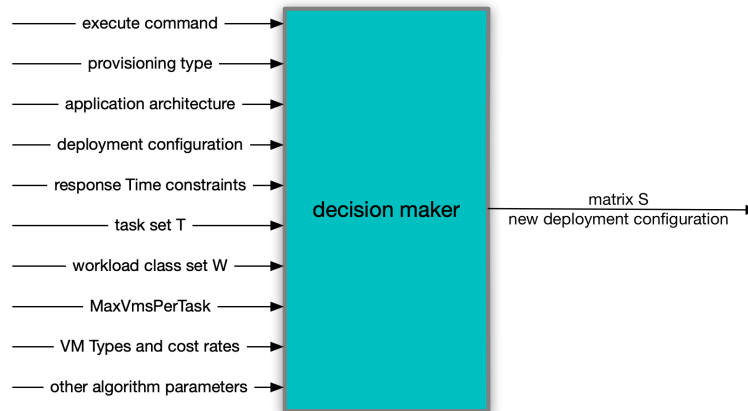


Figure 5.3: Decision maker block diagram

In dynamic provisioning, the [decision maker](#) controls the scaling of an application. Figure 5.3 shows a decision maker. A simple decision maker may replicate the application containers that are running under high system utilization to distribute the load to the new containers. A more sophisticated decision maker is capable of fine-grained decision making by having visibility into the tasks that compose the application and the interactions between those tasks. In such fine-grained decisions, the input to the decision maker is composed of, but not limited to the application workload, [application architecture](#), desired objectives and constraints. The result of the decision is an [deployment configuration](#), which is built of the number and type of VMs or containers for each application task. The chosen configuration is decided upon by considering the complete application and the desired result instead of considering individual VMs. This configuration meets the given objectives while considering the constraints, thereby allowing for an advanced decision making. The output deployment configuration is sent as an input to the [provisioning agent](#), so that the application's running configuration is adjusted to reflect the decision maker's decision.

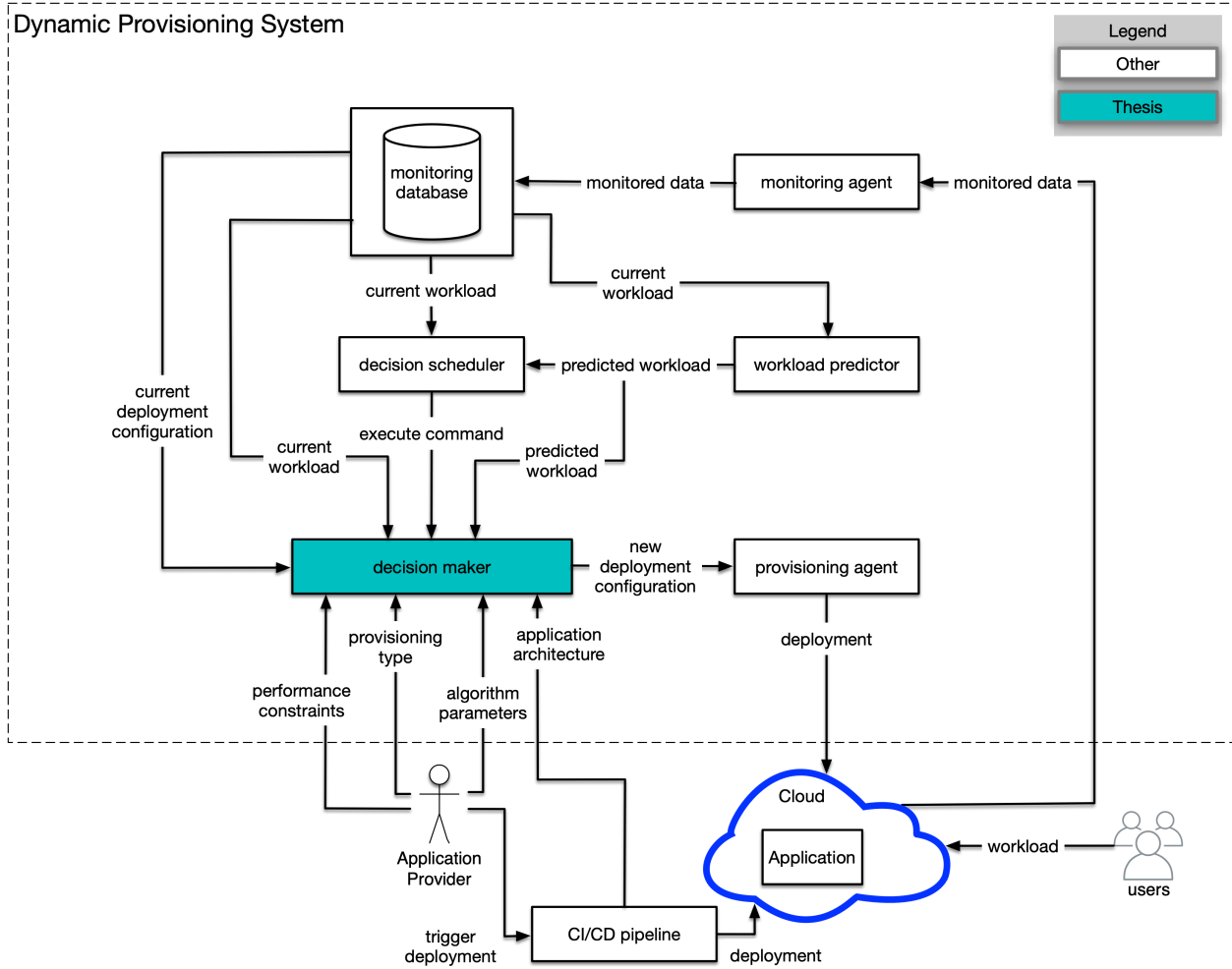


Figure 5.4: This figure shows the workings of a generic dynamic provisioning control system

### 5.2.2 Dynamic Provisioning Control System

Figure 5.4 shows how a generic dynamic provisioning control system works. It is a feedback loop system where the “[target system](#)” [128] is regulated through a [decision maker](#). The target system is represented by the application and its resources running within a cloud system. The decision maker is the controller component which runs the algorithms presented in this thesis.

The design and implementation of the decision maker is the focus of this thesis. The decision maker runs multiple algorithms search of a [deployment configuration](#) that meets the given performance constraints with minimal cost. Each application has its own decision maker, which is responsible for its scaling. Other components of the dynamic control system needs to exist before automated decision making can take place through the decision maker component.

In the following paragraphs, the stakeholders and the components of the dynamic provisioning control system are explained. The general workings of this system is also elaborated on.

**Application** AP deploys the application to the cloud through a [CI/CD pipeline](#). In this process, the [application architecture](#) is also uploaded to the [decision maker](#) for decision maker later.

**Workload** As the application begins to receive workload from the end-users, the decision maker regulates the system through the provisioning agent.

**Monitoring system** All observations are made through the [monitoring agent](#). This agent gathers workload data, and system and application health data and stores it in the monitoring database.

**Workload predictor** Using the time-series workload data from the monitoring database, the workload predictor forecasts the future workload. This forecast is for the next few time steps.

**Provisioning type and workload inputs** Two choices exists for the provisioning type input: [proactive provisioning](#) or [reactive provisioning](#). In the case of [proactive provisioning](#) the decision maker uses the predicted workload as input, whereas for [reactive provisioning](#) the current workload is used.

**Algorithm parameters** In this thesis, there are four algorithms choices made available by the decision maker: [ESA](#), [RSA](#), [SGAP](#) and [PGA](#). The algorithm parameters input, helps with deciding the algorithm to run. These parameters also include problem specific inputs such as those defined in Section [5.1.2](#), e.g. *MaxMsPerTask*, VM-Types, etc. Section [6.2](#) provides further details of the algorithm parameters input.

**Deployment configuration** The current [deployment configuration](#) is found through the monitoring. This configuration is a matrix of application tasks to VM-type allocations. e.g. an application with two tasks running on a cloud that has three VM-types would have, a deployment configuration consisting of  $2 \times 3 = 6$  elements, representing the allocation. This configuration is fed as an optional input into the decision maker. This input is optional because not all algorithms require this configuration for decision making.

**Application architecture** The [application architecture](#) represents the application tasks and their interactions, depicting the dependency amongst the tasks. The architecture could be represented by a directed graph where nodes represent the task with edges representing the number of calls between the tasks. The deployment configuration and application architecture together describe a deployed application. In this thesis, application architecture is represented through LQN models by extracting out the deployment configuration.

**Performance constraints** The performance constraints are sent as input through the [AP](#) in the form of mean response times of the workloads, and the decision maker works toward meeting these constraints.

**Decision scheduler** The decision scheduler decides when the decision making process begins.

**Decision maker and the execute command** After the execute command is sent from the decision scheduler, the [decision maker](#) runs the chosen algorithm and sends a new [deployment configuration](#) to the provisioning agent, who then applies changes to the application's configuration in the cloud, completing the control loop.

Except the decision maker, the other components in the dynamic provisioning control system are out-of-scope. Here, we assume an integration of the decision maker with all of the other components.

### 5.2.3 The decision making process

Figure [5.5](#) presented here follows from that discussion by explaining the internal workings of the decision maker, showing its block flow diagram.

The algorithm parameters input, helps with deciding the algorithm to run. On receiving the execute command from the decision scheduler, the decision maker begins running its decision making algorithm. In general, each algorithm performs the following actions to evaluate various deployment configurations searching for a candidate that solves the problem:

1. forms a temporary deployment configuration from all the possible configurations, given the workload (in user counts)

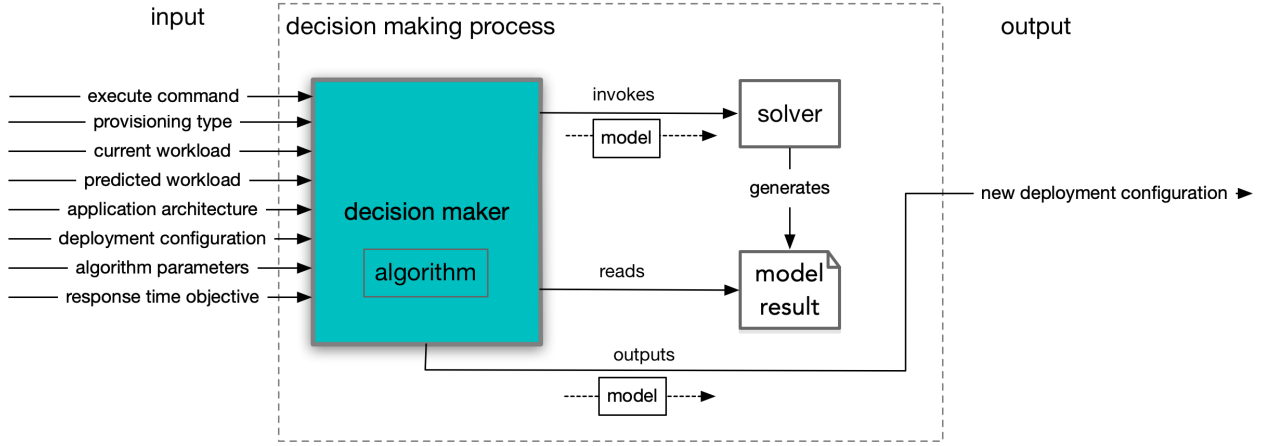


Figure 5.5: The decision making process

2. appends the deployment configuration to the application architecture (partial model) to form a complete performance model
3. sends the model for solving, by invoking the performance model solver
4. reads the model result
5. continues evaluating other deployment configurations based on the above process until the termination criteria has been reached, which includes the solution.

Each configuration is an allocation of the tasks to the VM-types. The number of configurations that the algorithm evaluates depends on the type of the algorithm. This will be explained in Section 5.3.

It is possible that the decision maker is unable to find a feasible and optimal solution. In such case, the algorithm terminates with no solution.

Once the decision maker has run successfully, it sends the solution (new deployment configuration) as an output to the provisioning agent. The provisioning agent would extract the deployment configuration from the model and deploys the changes to the application running in the cloud.

#### 5.2.4 Problem Encoding

Consider an application that has a five tasks, of which one is a reference task and two are non-replicable tasks. Therefore, there are a total of two scalable tasks which would make up the set  $T = \{T_1, T_2\}$ . These scalable tasks can be replicated.

This application is to be deployed to a cloud which provides three VM-types implying that  $v = 3$ .

The VM-types are:  $\{A, B, C\}$ .

Since there are  $2 \text{ tasks} \times 3 \text{ VMtypes} = 6 \text{ variables}$ , this example will be encoded as an Array of integers, with the length of array as six. Each element of the array represents the number of VMs of a VM-type allocated to a task. The first three elements indicate allocations of Task  $T_1$  to VM-types  $A$ ,  $B$  and  $C$  respectively. The second part of the array represents the allocations of Task  $T_2$  in a similar manner. An array  $\{3, 0, 1, 1, 0, 0\}$  indicates the following deployment configuration:

- $T_1$  is allocated to three  $A$  VM-type and one  $C$  VM-type. The starting index of this task is 0 and ending index is 2.
- $T_2$  is allocated to one  $A$  VM-type. The starting index of this task is 3 and ending index is 5.

### 5.2.5 Number of Evaluations

SGAP, RSA and PGA perform one evaluation per deployment configuration. The total number of evaluations are:

$$P \times I \tag{5.25}$$

The number of evaluations for ESA is presented in Section 5.3.1.

The fitness function is the *Cost* of the VMs (Equation 5.18), and the objective is to minimize the cost. For GA, each iteration of the algorithm also saves an elite (or best) individual and adds it back in to the next generation [129]. This process of selecting the elite individual for the next generation without altering it is known as “elitist selection or elitism” [49]. The cost of the elite individual is the lowest. When the constraints (Equations 5.19–5.24) are not met, then the evaluated solution is deemed infeasible. The algorithm terminates after  $I$  iterations.

### 5.2.6 Algorithm parameters

The following parameters need to be derived before running the algorithm in the decision maker:

1. Population size ( $P$ ) is required for PGA, RSA and SGAP
2.  $I$  is required for PGA, RSA and SGAP
3. Mutation Distribution Index (MDI) is required for PGA and SGAP
4. Cross-over Distribution Index (CDI) is required for PGA and SGAP



---

**Algorithm 5.1** RANDOMSEARCHALGORITHM (RSA). Based on Hadka [129]

---

**Input:** Algorithm parameters**Output:** Fittest individual from the last generation

```
1: do  
2:   Population = RANDOMGENERATION()  
3:   EVALUATION(POPULATION)  
4:   Fittest = FINDFITTEST(POPULATION  $\cup$  FITTEST)  
5: while termination-criteria-not-satisfied
```

---

The values of these parameters are based on the system under study. Section 6.2 explains how these parameters were derived for the case study presented in this thesis.

## 5.3 Decision Maker Algorithms

This section explains four algorithms of the decision maker. The main contribution of the thesis is PGA. The optimization problem referred to in this section is defined in Section 5.1.3.

### 5.3.1 Exhaustive Search Algorithm (ESA)

ESA evaluates all the deployment configurations searching for the best configuration that solves the optimization problem. There are a total of  $t \times v$  task-to-VM-type allocations, and for each allocation there are  $MaxVmsPerTask + 1$  choices available. The aforementioned choices for each allocation also include a task not being allocated to a VM-type (although the task could be allocated to another type). The zero deployment configuration, which has no tasks running on a VM is disregarded from the evaluations. Based on above, following are the total number of configurations evaluated by ESA:

$$(MaxVmsPerTask + 1)^{t \times v} - 1 \quad (5.26)$$

### 5.3.2 Random Search Algorithm (RSA)

Algorithm 5.1 shows a population based random search. The algorithm starts with generating a random population by calling the RANDOMGENERATION() function. This population is evaluated for their fitness by calling the EVALUATION() function. The fittest individual is selected from this population by calling the FINDFITTEST() function and saved to a variable. The next iteration of

the algorithm begins by generating a new random population and evaluating it. The next step is to find the fittest from the last fit individual and the new population. The algorithm continues until the number of iterations reach  $I$ , after which it terminates. The result of the algorithm is the fittest individual from all the randomly generated individuals.

### 5.3.3 Simple Genetic Algorithm for Performance (SGAP)

In Algorithm 2.1, a SGA was presented. The SGAP presented in this subsection and the PGA presented in the next (Section 5.3.4), both follow the same set of operations from SGA.

#### 5.3.3.1 Simulated Binary Crossover

Since the decision variables are not binary integers, a single-point binary crossover as mentioned in Section 2.4.5 cannot be applied. For this purpose we use the Simulated Binary Crossover (SBX) operator, which is made available by the MOEA framework. This operator simulates a single-point binary crossover for real-values [129]. The SBX operator was introduced by Deb and Agarwal [130, 131].

The SBX operator takes in two parameters as input [129]:

1. rate: probability of whether the operator would be applied
2. CDI: determines how near the offsprings are to their parents, with larger values creating more nearness.

#### 5.3.3.2 Polynomial Mutation

Similar to the crossover operation, a simple binary mutation operator cannot be used in our purposes. Here, Polynomial Mutation (PM) is used, which is also made available by the MOEA framework. This operator simulates a binary mutation for real-values [129]. The PM operator was introduced by Deb and Mayank [132].

The PM operator takes in two parameters as input [129]:

1. rate: probability of whether the operator would be applied
2. MDI: determines how near the offspring is to their parent, with larger values creating more nearness.

### 5.3.4 Performance-oriented Genetic Algorithm (PGA)

The **PGA** differs from **SGAP** in the crossover and mutation operators. Following is an explanation of both these operators.

#### 5.3.4.1 Layered Bottleneck Crossover

**PGA** uses **LBX** as the crossover operator. **LBX** is based on **layered bottlenecks**. In Section 2.3.3, details of layered bottlenecks was presented.

Algorithm 5.2 shows the pseudo-code of the operator. Following explains this code:

**Inputs and outputs** The operator takes in the following inputs: a pair of parents:  $p1, p2$ , the **SBX** parameters: *rate* and *CDI*, the workload set:  $W$ , and the response time constraints of all workloads:  $RTobj$ . After crossover has been performed on the parents, the output is a pair of children  $c1, c2$ .

**Lines 1–2** The algorithm begins by invoking the SOLVELQN API for both parents on their workloads. The API calls the LQNS solver, which is invoked only twice throughout this operator, once per each parent. The parents — and the individuals in general — represent the deployment configurations, and their configuration is included with the workload and the architecture of the system to form a complete LQN model. This complete model is solved by the solver. The result is parsed to retrieve the mean response times of all workloads for both parents.

**Lines 3–4** Once the response times have been found, the algorithm checks whether the constraints are met by each parent. If the constraints are met then the variables  $p1Valid$  or  $p2Valid$  corresponding to the parents is set to *True*, and *False* otherwise.

**Line 5 and Line 8** FINDMAXBSTRENGTHTASK API is invoked to determine which task of the parent has the highest *BStrength* value (bottleneck task), which is assigned to the variable  $b1$  and  $b2$ . In this case, the cached results from the previous LQNS run is used.

**Lines 6–7 and Lines 9–10** FINDCOST API is passed in two parameters: the parent and the task. Based on the task-to-VMtype allocation in the parent’s configuration, the API, finds the cost of all

---

**Algorithm 5.2** LAYERED BOTTLENECK CROSSOVER (LBX)

---

**Input:**  $p1, p2, W, RToj, rate, CDI$ **Output:**  $c1, c2$ 

```
1:  $\{RT_{W_i}(p1) \mid 1 \leq i \leq w\} = \text{SOLVELQN}(p1, W)$ 
2:  $\{RT_{W_i}(p2) \mid 1 \leq i \leq w\} = \text{SOLVELQN}(p2, W)$ 
3:  $p1Valid = \text{True if } \forall W_i, RT_{W_i}(p1) \leq RToj_{W_i}$ 
4:  $p2Valid = \text{True if } \forall W_i, RT_{W_i}(p2) \leq RToj_{W_i}$ 
5:  $b1 = \text{FINDMAXBSTRENGTHTASK}(p1)$ 
6:  $Cost_{b1,p1} = \text{FINDCOST}(b1, p1)$ 
7:  $Cost_{b1,p2} = \text{FINDCOST}(b1, p2)$ 
8:  $b2 = \text{FINDMAXBSTRENGTHTASK}(p2)$ 
9:  $Cost_{b2,p1} = \text{FINDCOST}(b2, p1)$ 
10:  $Cost_{b2,p2} = \text{FINDCOST}(b2, p2)$ 
11:  $\{c1, c2\} = \text{SBX}(p1, p2, CDI, rate)$ 
12:  $i = \text{STARTINDEXOF}(b1, p1)$ 
13:  $j = \text{ENDINDEXOF}(b1, p1)$ 
14:  $k = \text{STARTINDEXOF}(b2, p2)$ 
15:  $l = \text{ENDINDEXOF}(b2, p2)$ 
16: if  $p1Valid \ \&\& \ p2Valid$  then
17:   if  $Cost_{b1,p1} < Cost_{b1,p2}$  then
18:      $c2[i..j] = p1[i..j]$ 
19:   else
20:      $c1[i..j] = p2[i..j]$ 
21:   end if
22:   if  $Cost_{b2,p2} < Cost_{b2,p1}$  then
23:      $c1[k..l] = p1[k..l]$ 
24:   else
25:      $c2[k..l] = p2[k..l]$ 
26:   end if
27: else if  $p1Valid \ \&\& \ !p2Valid$  then
28:    $c2[i..j] = p1[i..j]$ 
29:    $c1[i..j] = p1[i..j]$ 
30: else if  $!p1Valid \ \&\& \ p2Valid$  then
31:    $c2[k..l] = p2[k..l]$ 
32:    $c1[k..l] = p2[k..l]$ 
33: else
34:   No changes made to  $\{c1, c2\}$ 
35: end if
36: return  $\{c1, c2\}$ 
```

---

VMs on which the task is running. This cost of running the task as per the parent's configuration, is assigned to one of the respective variables  $Cost_{b1,p1}$ ,  $Cost_{b1,p2}$ ,  $Cost_{b2,p1}$ , and  $Cost_{b2,p2}$ .

**Line 11** SBX API takes in four parameters as input: the two parents and the specific parameters for the operator  $CDI$  and  $rate$ . This runs the **SBX** operation on the parents producing two children

$c1$  and  $c2$  as outputs.

**Lines 12–15** `STARTINDEXOF` and `ENDINDEXOF` APIs take two parameters as input: task and the parent. These APIs determine the starting and ending index of the given task within the parent. The encoding of the algorithm has been specified in Section 5.2.4.

**Lines 16–26** If variables  $p1Valid$  and  $p2Valid$  are *True*, i.e. both parents  $p1$  and  $p2$  meet the response time constraints, then based on the cost of their bottleneck tasks, update the **genes** of the children  $c1$  and  $c2$ . Figure 5.6 explains the details of this crossover. The process begins by focusing on the indices corresponding to bottleneck  $b1$  within the parents and the children. The idea is if  $Cost_{b1,p1}$  is lesser than  $Cost_{b1,p2}$  then we have to pay lower to achieve higher utilization of the task, and therefore the genes at indices corresponding to  $b1$  in child  $c2$  are updated to the genes of  $b1$ . If this  $Cost_{b1,p1}$  is not lower than  $Cost_{b1,p2}$  then we update genes of child  $c1$  with genes of  $p2$  for the  $b1$  task. Similar process is followed for  $b2$  bottleneck.

**Lines 27–32** If of both parents only  $p1$  meets the response time constraints then the genes corresponding to bottleneck  $b1$  are transferred from  $p1$  to the children. This is done because parent  $p1$  generates a feasible deployment.

On the other hand, if only  $p2$  meets the constraints then the genes corresponding to bottleneck  $b2$  are transferred from  $p2$  to the children.

**Lines 33–35** If both parents are not able to meet the response time constraints then no changes are made to children  $c1$  and  $c2$ , and the SBX operation performed in Line 11 is the output.

The concept of the algorithm has been captured in Lines 16–26 and Figure 5.6. The idea here is to find high *BStrength* tasks in the parents and propagate their genes to the children if they have lower cost.

#### 5.3.4.2 Layered Bottleneck Mutation

PGA uses **LBM** as the mutation operator. Similar to the crossover operator, **LBM** is based on **layered bottlenecks**.

Algorithm 5.3 shows the pseudo-code of the operator. Following explains this code:

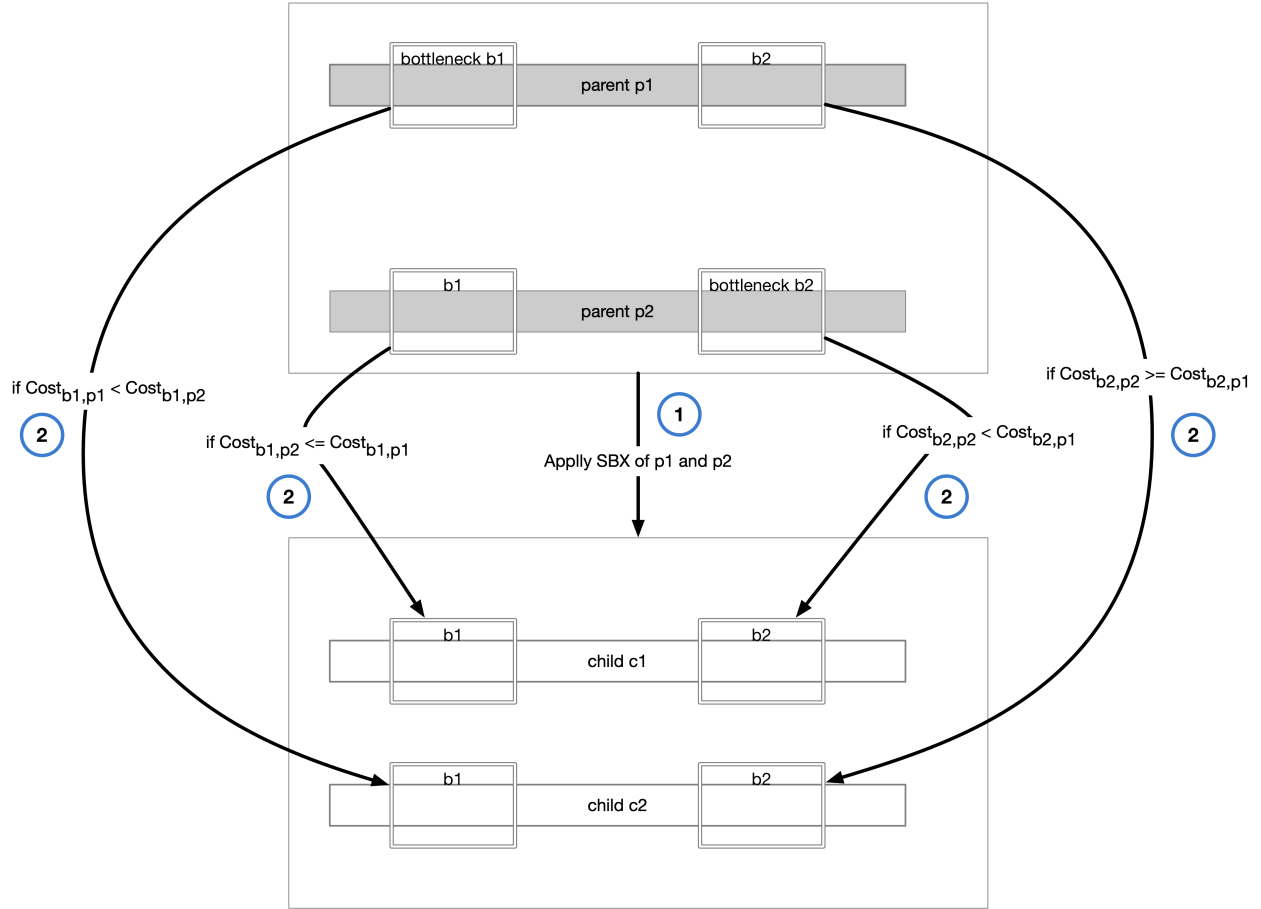


Figure 5.6: LBX when both parents meet response time constraints. First, SBX is applied to both parents which produces two children. Second, based on if-else logic, genes from parent are copied to the children.

**Inputs and outputs** The operator takes in the following inputs: a parent  $p1$ , the **PM** parameters:  $rate$  and  $MDI$ , the workload set:  $W$ , and the response time constraint of all workloads:  $RTobj$ . After mutation has been performed on the parents, the output is a child  $c1$ .

**Line 1** The algorithm begins by invoking the SOLVELQN API for the parent on its workloads. The API calls the LQNS solver, which is invoked only once throughout this operator. The result is parsed to retrieve the mean response times of all workloads for the parent.

**Line 2** Once the response time has been found, the algorithm checks whether the constraint is met by the parent. If the constraint is met then the variable  $p1Valid$  is set to *True*, and *False* otherwise.

---

**Algorithm 5.3** LAYERED BOTTLENECK MUTATION (LBM)

---

**Input:**  $p1, W, RTojb, rate, MDI$ **Output:**  $c1$ 

```
1:  $\{RT_{W_i}(p1) \mid 1 \leq i \leq w\} = \text{SOLVELQN}(p1, W)$ 
2:  $p1Valid = \text{True if } \forall W_i, RT_{W_i}(p1) \leq RTojb_{W_i}$ 
3:  $\text{CALCULATEBSTRENGTHOFTASKS}(p1)$ 
4:  $numVariables = t \times v$ 
5:  $c1 = \text{ARRAY}(\text{length} = numVariables, \text{default} = 0)$ 
6:  $weight = \text{ARRAY}(\text{length} = numVariables, \text{default} = 0)$ 
7:  $sum = 0$ 
8: if  $p1Valid$  then
9:   for  $i = 1 \dots numVariables$  do
10:     $vType = \text{VMTYPEAT}(i)$ 
11:     $task_{vType} = \text{TASKAT}(i)$ 
12:     $weight[i] = BStrength_{task_{vType}}$ 
13:     $sum = sum + weight[i]$ 
14:   end for
15:   for  $i = 1 \dots numVariables$  do
16:     $nextRandomNum = \text{NEXTRANDOMNUMBER}(0,1)$ 
17:    if  $nextRandomNum \leq (1 - (weight[i] \div sum))$  then
18:       $c1[i] = \text{PM}(p1[i], MDI, rate)$ 
19:    end if
20:   end for
21: else
22:    $c1 = \text{PM}(p1, MDI, rate)$ 
23: end if
24: return  $c1$ 
```

---

**Line 3** CALCULATEBSTRENGTHOFTASKS API is invoked to calculate the  $BStrength$  value of each task of the parent. The value is stored as part of each task.

**Lines 4–7** NUMVARIABLES represents the number of variables of the problem and is calculated as a product of  $t$  and  $v$ . The decision variables are represented by Matrix  $S$  (Equation 5.9).

The ARRAY API creates a new array by taking in two parameters:  $length$  of the array and the  $default$  value of each element.

Using the ARRAY API, two variables  $c1$  and  $weight$  are created.  $c1$  array represents the child and is the eventual result of the algorithm. The  $weight$  array is used in determining the probability of PM operation being applied for each variable.

The  $sum$  variable is a sum of all weights and along with the  $weight$  array is used for determining the probability of applying PM operation.

**Lines 9–14** *weight* array is computed for each variable. For each variable, the task and the VM-Type is found by using two APIs respectively: VMTypeAT and TaskAT. Based on these two variables, the *BStrength* of the task running on the particular VM-Type is assigned to the weight. The *sum* is calculated by adding the elements of the *weight* array.

**Lines 15–20** For each variable, it is determined if PM operation is to be performed. *nextRandomNum* stores a random number between 0 and 1. If this random number is less than and equal to  $(1 - (weight[i] \div sum))$  then PM API is called to perform mutation on the variable.

**Lines 8–24** If the parent meets the response time constraints then we perform the layered bottleneck mutation on each variable by calling the PM API on each *gene* (Line 18), otherwise we perform the default PM operation by calling PM API on the whole parent (Line 22).

The concept of the algorithm has been captured in Lines 8–24. The idea here is to have lower probability of mutation with higher *BStrength* value considering that the parent already provides a feasible solution by meeting the response time constraint.

## 5.4 Discussions

In this section, the discussions associated with the solutions are presented. Following is the outline of this section. Section 5.4.1 compares and contrasts SatisfyQoS and PGA algorithms. Section 5.4.2, explains the reasons for choosing GA for problem solving. Section 5.4.3 lists the limitations of the work presented in this chapter. Section 5.4.4 suggests improvements on the decision maker's architecture. Section 5.4.5 discusses measures that can be taken to handle decision maker failures. Section 5.4.6 explains the importance of timely decision making. Section 5.4.8 mentions how monitoring and alerting can further help with decision making. Section 5.4.9 outlines how a new decision maker version can be released. Section 5.4.10 suggests enforcing cost and security around the decision maker. Section 5.4.11 discusses how performance of the decision maker can be improved.



### 5.4.1 Similarities and Differences: SatisfyQoS and PGA

In the previous chapter, [SatisfyQoS](#) algorithm was presented. Both [SatisfyQoS](#) and [PGA](#) have similarities and differences. Both are based on bottleneck theory and their common goal is to meet performance constraints.

In comparison to [SatisfyQoS](#), [PGA](#) adds the minimization of cost as an objective, where VM-types with different cost rates are used. [PGA](#) does not require a deployment configuration and provides a more complete solution, where resources can be added and removed. Furthermore, [PGA](#) does not require the  $sat_{thresh}$ . [SatisfyQoS](#) relies on finding layered bottlenecks, for which  $sat_{thresh}$  is used as an input, whereas, [PGA](#) uses [BStrength](#) metric. [SatisfyQoS](#) does not facilitate the removal of resources.

The Task-to-VM relationship is also different between the two solutions. [SatisfyQoS](#) assumes a many-to-many relationship, where [PGA](#) assumes and enforces a one-to-many relationship between tasks and VMs. This is seen from [Figure 4.1](#) and [Figure 5.1](#).

The strength of [SatisfyQoS](#) is in making minimal changes to the application when meeting the performance constraints.

### 5.4.2 Choosing genetic algorithm

There were multiple reasons for choosing genetic algorithms for solving the dynamic provisioning problem presented in this thesis.

Section [2.4](#) highlighted the differences between [GA](#) and greedy search algorithms such as Dijkstra and A\*, stating that for the latter greedy algorithms, the path to the goal state has to be minimal. For the problem we are solving, the cost of the path to the goal state is not important, but instead finding the goal state is. Also for Dijkstra and A\*, the end state is already known, whereas for the problem we are solving, the end state is to be found.

For the problem stated in [Section 5.1.3](#), heuristics are well-suited in finding near-optimal solutions. [Chapter 4](#) demonstrated how bottleneck analysis helps with decision making in provisioning. This heuristic is perfect to be used within [GA](#), to guide the complete decision making process.

Instead of searching for the next neighbour and finding a local-minima (hill-climbing), or using single-solution approach (simulated-annealing), a populated-based global-search approach ([GA](#))

have been favoured for solving the dynamic provisioning problem.

### 5.4.3 Limitations

This section outlines some of the limitations of [PGA](#).

#### 5.4.3.1 Tail distribution

One of the limitations of the proposed solution in this thesis is the use of mean response times for decision making. This approach does not work well for response times that have a long-tail distribution. A better approach will be solving the problem where 95% of the response time results meet the given constraint.

#### 5.4.3.2 Flash crowds

The current solution would not be able to accommodate a sudden workload increase. This phenomenon is known as a “flash crowd” [\[133\]](#). Through workload forecasting, which is employed within proactive provisioning, the workload in the next few time steps can be predicted. However, in the case of these flash crowds, the workload is higher than what is normally expected.

Keeping spare VMs can help in this situation. These extra VMs can be kept running un-utilized and anytime the workload is higher than the provisioned capacity, the traffic can be forwarded to them.

Monitoring for flash crowds and automatically provisioning additional resources can also help alleviate the stress induced by flash crowds on the application.

Admission control also comes to rescue in this case, with limiting excess request rates to the application. Without admission control, the application is exposed to service downtime on excess requests and denial-of-service attacks. An example of such admission control is seen when using Google Sheets API [\[134\]](#).

### 5.4.4 Architecture

This section discusses ways in which the architecture of the decision maker can be improved.

#### 5.4.4.1 Separating implementation from configuration

The input parameters into the decision maker are case study specific. The number of workloads and response time constraints and other parameters are dependant on the application under study. Based on the current implementation, which is outlined in Appendix B, the decision maker is very much coupled with these parameters and therefore for each new application a new decision maker implementation (with minor changes) will be required. This is not ideal. A better approach would be to separate the decision maker software from the input configuration. In this approach, there will be one decision maker software but multiple configurations for each application. Here, when a new application needs to use the decision maker then a general available release of the decision maker software would be deployed for the application and the configuration would be application specific.

#### 5.4.4.2 A shared decision maker

Instead of having multiple deployments of the decision maker, i.e. one deployment per application (as suggested in Section 5.4.4.1), another alternative would be having a shared decision maker service. The benefit of this approach is centralized management of decision making with the APs making API calls to one service.

Both designs — the design presented in this subsection and the design presented in Section 5.4.4.1 — would need to be evaluated before implementation. A centralized management creates maintenance and operational overhead for the one team that owns the decision maker, whereas, a per application decision maker moves the operational responsibility to the APs.

#### 5.4.4.3 Decision-maker-as-a-Service

From a shared decision maker, we can further improve the application architecture of the decision maker. The decision maker can be refactored into smaller and reusable services by adopting a microservice architecture. Following this approach, we can have a minimum of the following three services:

1. VM-type service: This service will return the types of VM available

2. Cost service: Given a deployment configuration of different VM-types, the Cost Service would be able to return back the cost
3. Performance Model service: This service will be used for working with the performance model.
4. Decision maker service : This core service will handle the decision making and use the other services mentioned above.

The major benefit of the above application architecture, other than reusability, is that there is separation of concerns and troubleshooting is easier.

#### **5.4.5 Handling failures**

The decision maker is a critical component in the dynamic provisioning control system. A fault in the decision maker could trigger a failure. In such situations, the complete system is brought to a halt. For this purpose, the complete system has to incorporate fault-tolerance, self-healing, monitoring, logging, tracing, alerting and other mechanisms to help bring the system back to operational.

In a microservice architecture, retry-logic for API calls between services would address intermittent failures. Caching would also aid here where the results are the same and the system state has not changed. Redundant services and data replication also saves the system from failures. When one healthy service is down then the redundant service is still active and the system still remains functional.

Usually with retry-logic, the caller service waits for a short time duration before retrying the API call again. This short wait is also useful when self-healing is in place. The callee service can quickly seal-heal if any repair is required.

System and service health monitoring, logging, tracing and alerting, aid the operators in bringing the system back up by diagnosing the issues quickly. With a microservice architecture, the diagnosis of the issue may be even faster, as the health of the microservices can indicate which particular service is failing and causing the upstream services to fail as a result.

In a distributed system, failures are expected, and they cannot be avoided. The goal here is to reduce metrics such as mean-time-between-failures and mean-time-to-repair. On the other hand, the goal is to over time decrease the mean-time-to-discovery.

### 5.4.6 Timeliness

Decision making can be a time consuming process. The total time from the point the decision scheduler invokes decision maker to the point the decisions are completely actioned on the cloud has to be kept to a minimum. Here, this time duration is referred to as decision-to-provision time. This is important in both reactive and proactive decision making. In the case of reactive provisioning, the application is already seeing the workload and needs resources quickly. In proactive decision making, the forecasted workload is for a particular time  $t$  in the future. If the decision-to-provision time is long and does not happen before  $t$  then the application will not have enough resources to meet the workload at time  $t$ . In the following subsection (Section 5.4.7), we explain how a hybrid policy can address these concerns.

### 5.4.7 Hybrid policy

For timely decision making we can employ a hybrid policy, where our analytical model based policy would be combined with a rule-based policy in the decision maker. In the case where our analytical model based policy is unable to derive a decision on time, the rule-based policy will be used for quick decision making.

### 5.4.8 Monitoring and alerting

Monitoring and alerting can help when the application is under-provisioned. This under-provisioning could either be because of poor decision making by the decision maker or because of inability of the system to deliver the decision result on time. On seeing the alert, operators can manually provision extra resources for the application. In the opposite situation, when the application has too many resources allocated to it and is over-provisioned, the operators should manually decrease resource count.

For instances when the decision maker is unable to solve the given problem, then the monitoring and alerting system can inform the operators to manually make decisions and provision adequate resources.

#### **5.4.9 Deploying new versions of the decision maker**

Enhancements and bug fixing to the decision maker software should follow a systematic process. By setting up a [CI/CD pipeline](#), the changes would be properly managed.

#### **5.4.10 Enforcing cost and security of the decision maker**

The decisions made by the decision maker improve the performance and control the cost of the running application. Bad decisions would do the reverse. Bad decisions may arise due to a bug in the software or due to the service being compromised. An oversight into the decision making is necessary for this purpose.

The records of the decisions made should be audited to check the accuracy of decision making. This could also be achieved through an automated oversight service with rules in place for the audit. A cost threshold as a rule in the oversight service would help ensure costs do not exceed the threshold.

#### **5.4.11 Performance**

This section suggests how performance of the decision maker can be improved.

##### **5.4.11.1 Profiling of the app**

In Section [5.4.6](#), a particular attention was given to the timeliness in decision making. For this purpose, understanding the software and based on this improving the performance is essential. To gain better understanding of the software and bottlenecks, profiling the decision maker is required. This is a future work.

##### **5.4.11.2 Caching of LQN model results**

In Section [5.4.5](#) caching was mentioned about context of handling failures. Caching can also be used for improving performance. The current implementation of the decision maker uses caching for LQN model results.

#### 5.4.11.3 Offline search

By running [ESA](#) offline, a complete database of optimal deployment configurations for different workloads can be created. Although this eliminates the need for an online running algorithm such as [PGA](#), this approach may be useful in cases where quick decision making is required.

## 5.5 Conclusions

This chapter begins by proposing a VM provisioning problem that considers performance and cost in decision making. The assumptions that have been held in formulating the problem are presented first, followed by notations and a formal problem definition. The presented solution is the decision maker which runs multiple algorithms. The PGA is the main contribution of this thesis. This solution use analytical performance models for their decisions.

Along with [PGA](#), the following three algorithms are presented for comparison purposes: [ESA](#), [RSA](#) and [SGAP](#). ESA navigates through the complete search space and finds an optimal configuration. RSA generates random solutions and chooses the best configuration as the solution. PGA differs from SGAP in the use of bottleneck theory for crossover and mutation operations. These three algorithms are introduced for comparison purposes, for determining the optimality of solutions generated by PGA, and for evaluating the performance of PGA. In Chapter 6 a case study is presented that evaluates these algorithms.

This chapter wraps up with discussions on various topics related to the decision maker. Discussion topics include limitations, architecture improvements, fault-tolerance, performance improvements and more.

# Chapter 6

## Case study: Bike routes application

In this chapter, we evaluate the different algorithms of the decision maker presented in Chapter 5 on a bike routes web application. Parts of this chapter have been accepted for publication in [124].

**Chapter Outline** Section 6.1 presents an overview of the application in study. The topics in the section include application's process flow, measurements and performance model. Section 6.2 explains the input parameters for the decision maker, based on the case study. Section 6.3 presents the results of running the decision maker on the application for different user workloads. Section 6.4 presents the conclusions.

### 6.1 Bike routes web application

In this section, the web application under study, MyBikeRoutes-OSM is presented. In 2011, the application was decommissioned, however we have data from [22], which is used for the case study. After introducing the web application, the Process-Flow Diagram is presented.

#### 6.1.1 Overview

**MyBikeRoutes-OSM** MyBikeRoutes-OSM, is a repository of bicycle routes that allows the users to create and share them from anywhere in the world. Users can search for the best bike route between given source and destination locations, i.e. best path search.

**Best path search** The best path search functionality is a prototype where the search is essentially performed using the roadways instead of the bicycle routes, i.e. although the bike routes can be



added and displayed through the Browser, they are not used in the search. This however does not affect the study as our main focus is towards deriving information about the system's performance.

**Map** The application uses OpenLayers JavaScript API to display OpenStreetMap (OSM) maps [135]. In the particular case of this application, the maps are rendered on the Browser through an ordering of pre-rendered map tiles/images residing on the server. When a user visits the site, the server sends the pre-rendered map titles back to the Browser to display the map.

**Database** PostgreSQL database functions as the datastore of the bike routes data and provides best path routing features, where the routing functionality is made available by the pgRouting project [136].

**Apache-PHP server** For displaying bike routes data or for route-search, OpenLayers API at the client-side communicates with the PostgreSQL database through an Apache-PHP server.

### 6.1.2 LQN Performance Model

This section explains the model structure, the input parameters and the complete model that is used in the case study.

#### 6.1.2.1 Base-Scenario Model Structure

**Browser reference tasks** *Browsers* represent the reference task which initiates the requests to downstream tasks. These browsers are modeled as infinite servers, running on *pClient*. This is a closed model where once a request session is completely processed, the customer is sent back to begin a new set of requests after waiting for a given think time.

**AppServer task** Each *Browser* sends requests to the entries of the *AppServer* task. These entries represent the different requests made to the server. *sendHTML*, *sendJS1*, *sendJS2*, *sendJS3* and *viewRoutes*, altogether are the requests sent on visiting the website. *routing1* and *add1*, are the requests sent for the first best path search. Similarly, *routing2* and *add2* are requests sent for the second best path search.

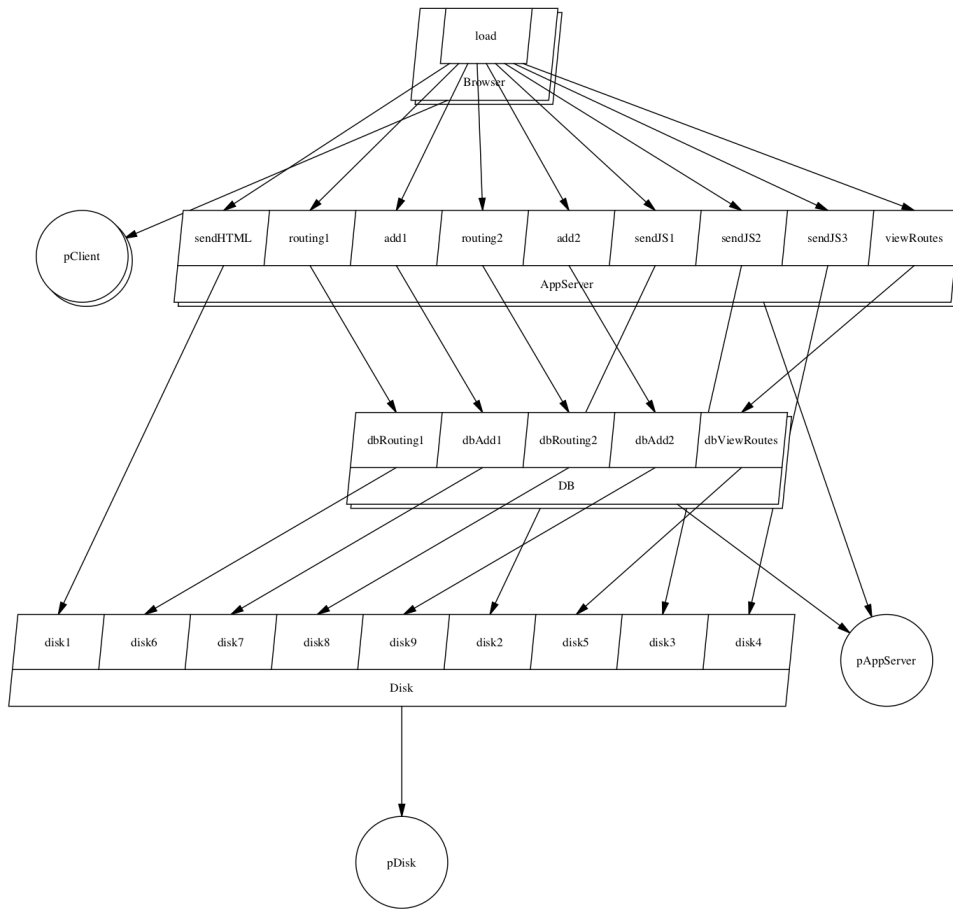


Figure 6.1: Base-Scenario LQN Model with no input parameters

**DB task** The entries of the *DB* task represent the requests from the *AppServer* entries to the database. On visiting the website the *dbViewRoutes* entry of the *DB* task is called by the *viewRoutes* entry of the *appServer* task. *dbRouting1* and *dbAdd1* are called for the first best path search. Similarly, *dbRouting2* and *dbAdd2* are called for the second best path search.

**AppServer and DB processor** Both *AppServer* and *DB* tasks execute on the same uni-processor (*pApache*), which has Processor Sharing (PS) scheduling discipline.

**Networks and activities** Although the model depicted in [22], included the network tasks and activities, both of these have been excluded to simplify the model.

**Disk** The disk is modeled by the *Disk* task, which has nine entries. Each entry relates to the nine requests issued by the *Browser*, i.e the first request (*reqHTML*) has its disk service demand provided by *disk1* entry, then *reqJS1* has the disk service demand provided by *disk2*, and following the same pattern for other requests.

**Requests** There are five request classes that require database access, i.e. *reqViewRoutes*, *reqROUTING1*, *reqROUTING2*, *reqADD1* and *reqADD2*. There is one SQL query executed for *reqViewRoutes*, which just involves retrieving the routes data. Both the routing requests (*reqROUTING1* and *reqROUTING2*) require running three queries, relating to the start and end points and finally the routing search. There is one query executed to insert a route for *reqADD1* and *reqADD2* requests. However, as mentioned in the earlier paragraph, each entry of *DB* task has only one visit made to it in the model, i.e. *dbViewRoutes*, *dbRouting1*, *dbAdd1*, *dbRouting2* and *dbAdd2* have only one visit from the upper *AppServer* task layer. Based on this, considering *dbViewRoutes*, the service times of each SQL query executed for *dbViewRoutes* was summed and presented finally as the service time of the *dbViewRoutes* entry. Similarly, service times for other entries of the *DB* task have been found. In the following paragraph, the sequence of execution that the model represents is explained in detail.

**The complete session** In the model, the *load* entry initiates the *Browser* requests to each entry of the *AppServer*, causing the execution of *sendHTML* to *viewRoutes*, representing a complete session. Any disk operations happens through the entries of the *DB* task, which run on *pDisk*. Some *AppServer* entries need to read or update data on the *DB*, forming a nested interaction, where the *AppServer* only sends reply back to the *Browser* when the *AppServer's* request has been responded back by the *DB*. The service demands for the *Base-Scenario* model are discovered by applying the *Utilization Law*, details of which are presented in the following section.

#### 6.1.2.2 Model input parameters

**Browser task** Since the load test did not include processing at the client machine because of browser rendering or page generation, the service demands for the *Browser* entries in the model are set to 0. The think time is set to 7 seconds.

**AppServer task** Based on the *MaxClients* [137] directive of Apache server, which sets a limit to the maximum number of processes that are available to handle client requests concurrently, the *AppServer* multiplicity is set to 150.

**DB task** Similarly, based on *max\_connections* [138] parameter of PostgreSQL database, which specifies the maximum concurrent connections to PostgreSQL database, the *DB* task multiplicity is set to 100.

**Disk** An assumption regarding *Disk* entries has been made in the model. For a request, only one interaction with the *Disk* entries is assumed, i.e. if both the *AppServer* and *DB* tasks perform certain number of disk I/Os for a particular request, then only one call to the *Disk* at the end of the nested interaction is depicted in the model. In this case, the service time of the respective *Disk* entry is taken as the product of number of disk I/Os and the average service time at the disk.

### 6.1.2.3 The case study model

Request class	AppServer (ms)	DB (ms)	Disk (ms)
reqHTML	9.62	-	0.01
reqJS1	0.85	-	0.01
reqJS2	4.8	-	0.01
reqJS3	0.64	-	0.02
reqViewRoutes	95.55	29.38	0.06
reqROUTING1	211.67	252.72	0.28
reqADD1	53.42	0.43	0.32
reqROUTING2	186.16	52.09	0.23
reqADD2	53.43	0.41	0.33

Table 6.1: Service Demand Parameters for Base-Scenario Model

The service times for each entry of the *Base-Scenario* model are shown in Table 6.1. Here, the *reqHTML* service time on the *AppServer* is 9.62 ms, which corresponds to the *sendHTML* entry service time. Similarly, *reqHTML* service time on *Disk* is 0.01ms, corresponding to the *disk1* entry.

Figure 6.1 depicted an LQN model with no input parameters. After having derived the input parameters, the complete LQN model for the case study is presented here. Figure 6.2 shows the base model that will used for the case study.

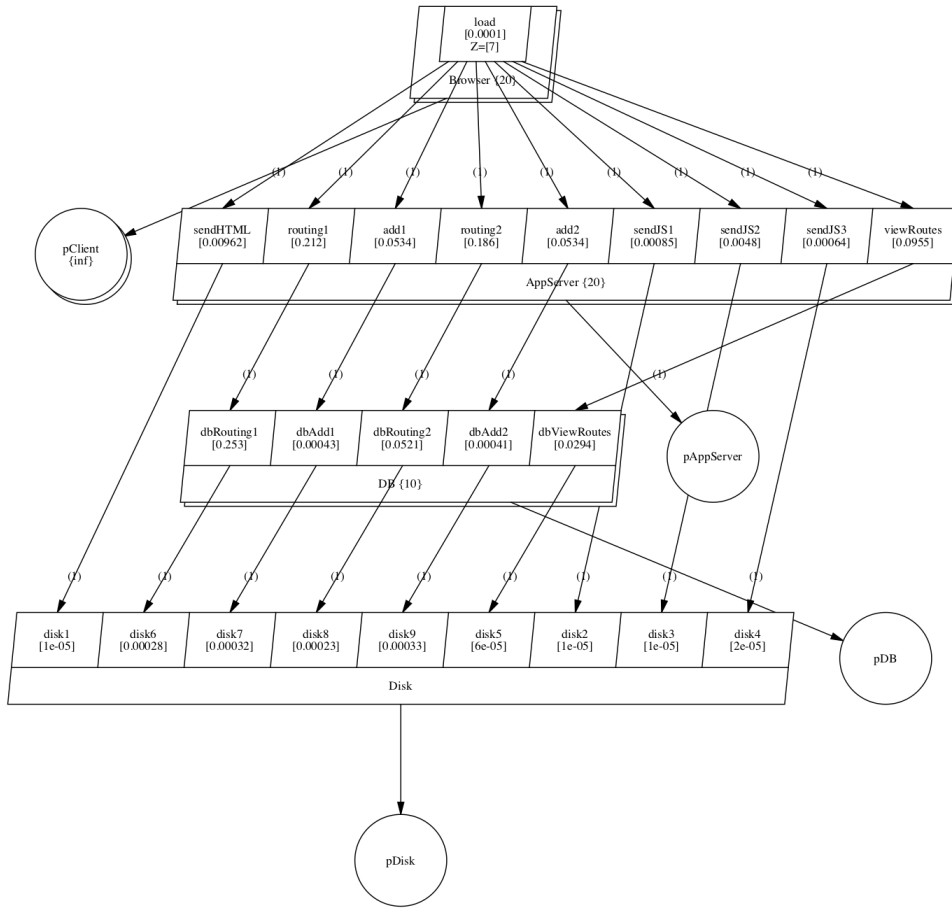


Figure 6.2: Case Study LQN Model

**Workload** In this particular instance, the number of Browsers are set to 20. This count will change as the workload for the decision making changes.

**VM types** The processors *pAppServer*, *pDisk* and *pDB* all are shown to be running on a uniprocessor. However, this completely depends on the choice of VM types offered by the CP and also on the deployment configuration that is being evaluated.

**Differences** This model in Figure 6.2 that we have used for the case study defers from the model presented in [22] in the following ways:

1. Threads of task *DB* is set to 10 instead of 150.
2. Threads of task *AppServer* is set to 20 instead of 100.
3. Network resources and activities are not included in the model. This has been explained in

### Section 6.1.2.1.

The reason for using lower thread counts is to allow to these tasks to be bottlenecks for which compute instances would be allocated when running the dynamic provisioning algorithms presented in thesis.

## 6.2 Decision Maker Inputs

Here, how the following parameters were derived for the case study is explained:

1. response time constraints
2. workload
3. algorithm parameters:
  - (a) VM-types and cost rates
  - (b) algorithm type
  - (c)  $MaxVmsPerTask$
  - (d)  $P$
  - (e)  $I$
  - (f)  $MDI$
  - (g)  $CDI$
  - (h) number-of-runs
  - (i) random-seed

The response time constraint is set to 8 seconds, which includes the think time of 7 seconds.

There is only one entry of the Browser task and therefore only one workload value for the number of users of this task is sent to the decision maker.

The VM types provided by the CP are shown in Table 6.2. The prices reflect the pricing for Amazon EC2 on-demand instances for Windows OS within Canada-central in circa 2017.

VM-Type	Processors	Cost (cents-per-hour)
small	2	20.3
medium	4	40.6
large	8	81.2

Table 6.2: VM Types

The algorithm type is one of the following: [ESA](#), [SGAP](#), [RSA](#), [PGA](#).

For selecting the values of  $P$ ,  $I$ ,  $MDI$  and  $CDI$ , the **PGA** algorithm was run for the parameter values specified in Table 6.3. Experiments are run by keeping all parameters constant except one and varying that parameter for a given range, and then moving on to varying another parameter.

Parameter	Range	Interval
$P$	10–60	5
$I$	5–35	5
$MDI$	10–30	5
$CDI$	5–25	5

Table 6.3: Parameters

Each experiment — for given parameter inputs — was run 20 times (number-of-runs) with number of users set to 20. The reported cost for the algorithm then is taken as the mean of all feasible runs for one complete experiment. The results of these experiments and the selected values are explained in the next section (Section 6.3.1) and shown in Table 6.5. The hardware and software configuration of the machine on which the experiments were run is shown in Table 6.4

Resource	Configuration
Operating System	16.04 LTS
Memory	5.7 GiB
Processor	Intel Core i5-2410M CPU @ 2.30GHz x 4
OS type	64-bit
Java	JDK 1.8
LQNS	5.10

Table 6.4: Case Study: Hardware and Software Configuration

The distinction between  $I$  and number-of-runs is that the **PGA** has not reached a termination state until we have reached the iteration count equal to  $I$ , and once this condition is satisfied then we have completed one run and generated a solution. Further runs in the experiment are executed for statistical purposes.

For generating random numbers, a seed value (random-seed) is set and static methods made available through the **PRNG** class provided by **MOEA** framework is used [129]. Setting this seed value allows for randomness between each of the 20 runs within one experiment, but helps with generating consistently same results when a new experiment is started with the same inputs.

For a given number of users, the experiments were performed for  $MaxVmsPerTask$  between 2–5. The *rate* for **SBX** is set to 1.0 and for **PM** is set to  $1 \div (t \times v)$ , which are the default rates used

within the MOEA framework.

## 6.3 Results and analysis

This section presents the results of running the decision maker with the bike routes application performance model as an input for various workloads. In total, 24 experiments ( $6 \text{ users} \times 4 \text{ MaxVmsPerTask}$ ), were performed. Each experiment was run 20 times for PGA, RSA and SGAP to calculate mean cost and mean runtime. ESA is only run once.

Section 6.3.1, presents the impact analysis results for parameters  $P$ ,  $I$ ,  $MDI$  and  $CDI$ . Section 6.3.2 showcases the end result of a solution. The section presents the optimal result for 220 users, allowing the readers to visualize the solution. The result shows a complete model including the deployment configuration and application architecture. Section 6.3.3– 6.3.5, presents the results for cost versus iterations, cost versus users, and runtime versus users. Appendix C includes raw data and extra graphs from the case study.

### 6.3.1 Impact of parameters

Figure 6.3 presents the impact of each of the parameters by showing how varying one parameter and keeping the others constant affects the mean cost of the VMs. As previously outlined in Section 6.2, the impact of the parameters was found by running experiments for 20 users. The result of the analysis was used in selecting the parameter values, which are shown in Table 6.5.

Parameter	Value
$P$	20
$I$	25
$MDI$	10
$CDI$	15

Table 6.5: Selected values for the Parameters

As population size increases, the mean cost rate of the 20 runs approaches and meets the optimum cost rate of 81.2 cents/hour. Optimum cost is reached for population sizes of 45, 55 and 60. However, this parameter is directly proportional to the number of evaluations (Equation 5.25), and therefore on the runtime. Considering this, 20 was chosen as the default value because it is not a high population size, and generates a near-optimal solution. Furthermore, the next improvement



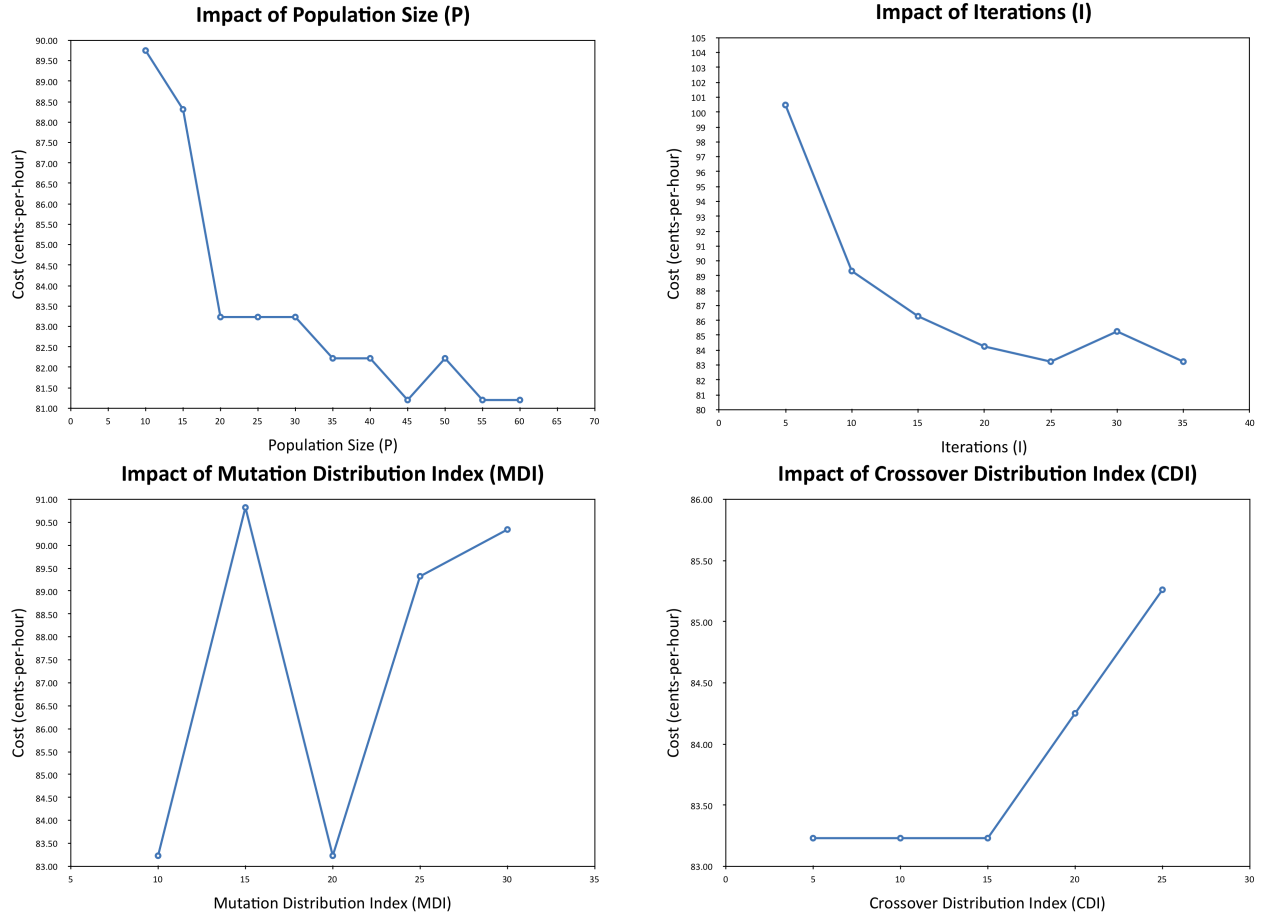


Figure 6.3: Impact of parameters - PGA algorithm. Only one parameters is modified where other parameters are kept constant. The default values of other parameters are shown in Table 6.5

after 20 comes from a population size of 35, which will amount to 1.75 times more evaluations, and is therefore disregarded.

The total number of iterations also is directly proportional to the number of evaluations and runtime. 25 iterations was chosen as the default as this value provided the best results with fewer iterations.

For MDI and CDI, the chosen values were 10 and 15, respectively, as those values provided the best results. When we select  $MDI = 10$ ,  $P = 20$  and  $I = 25$ , and then traverse through CDI values, it is seen that values 5 and 10 result in the same cost as value 15. However, for CDI, when values 5 and 10 were chosen and other graphs were re-evaluated then poor performance was seen for other parameters, and hence CDI was finally selected as 15.

Here, we present one optimal configuration that was found by running the ESA. Figure 6.4 shows the optimal configuration for 220 Users, where  $MaxVmsPerTask$  is set to 3. The response time of the optimal configuration is just under 8 seconds. The configuration is  $\{0, 0, 3, 0, 3, 0\}$ . Following the encoding explained in Section 5.2.4, we see that this depicts 3 large VMs for the first task (AppServer) and 3 medium VMs for the second task (DB).

For the optimality evaluation, 20 runs were executed for all combinations of users and  $MaxVmsPerTask$  for **PGA**, **RSA** and **SGAP**. **ESA** was run once for each combination. As the number of runs in each case were less than 30, t-distribution was used rather than the standard normal distribution for determining the 95% confidence intervals.

101

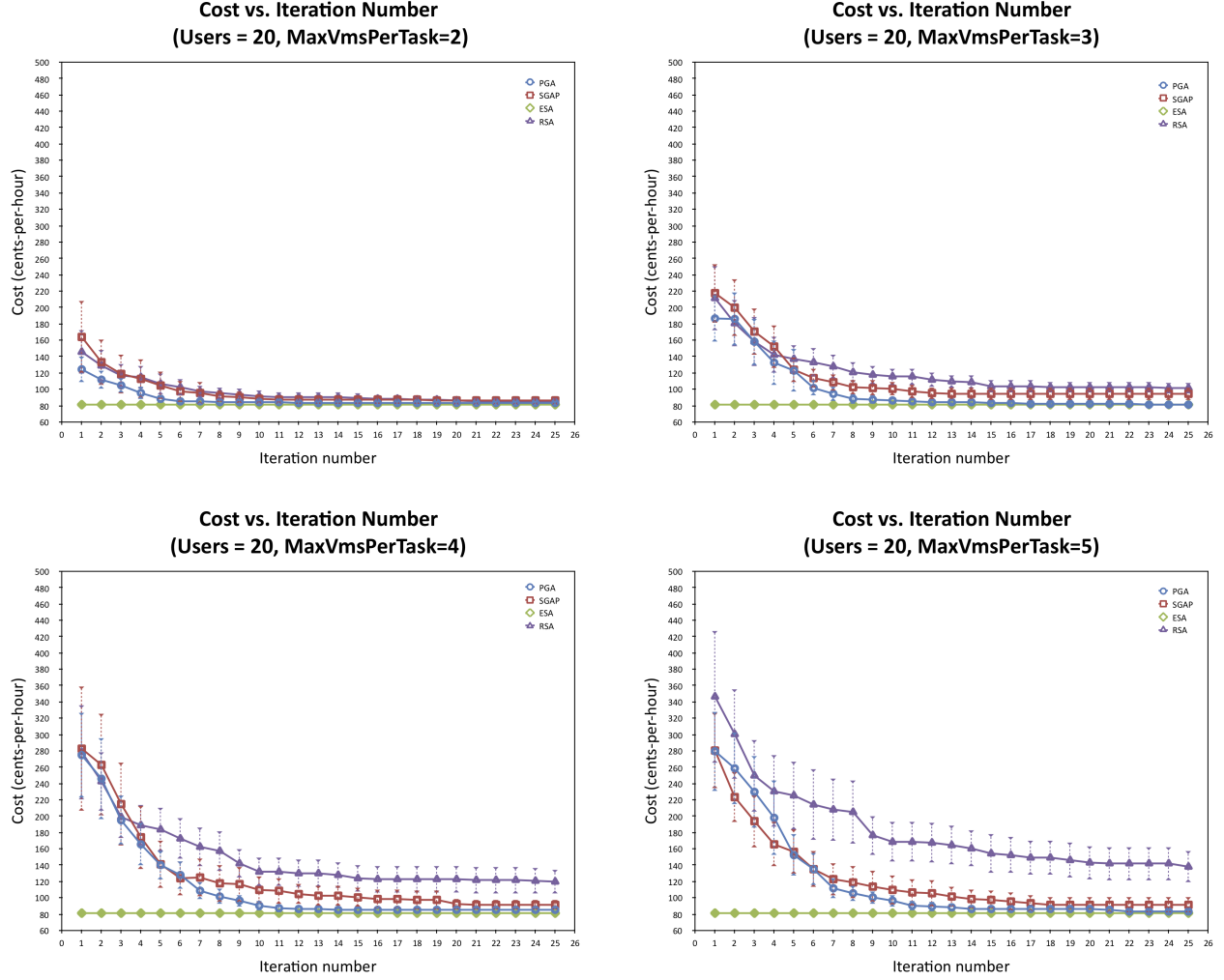


Figure 6.5: Cost vs. Iterations (Users=20)

is large, the search space is also large and the algorithms end iteration one with higher mean cost compared to when  $MaxVmPerTask$  is set to lower values. Overall, RSA performs worse than SGAP and PGA. Similar results are seen for other user counts, which is provided in Appendix C.

#### 6.3.4 Cost vs. Users

Figure 6.7 shows the Cost vs. Users graphs for  $MaxVmsPerTask$  between 2–5. Results from 23 experiments are shown because no feasible solution was found for  $MaxVmsPerTask = 2$  and Users=220, as no configuration meets the response time constraint.

Across each graph, as  $MaxVmsPerTask$  increases, we see that the number of feasible runs increase. This is because the search space increases with increase in  $MaxVmsPerTask$  and the

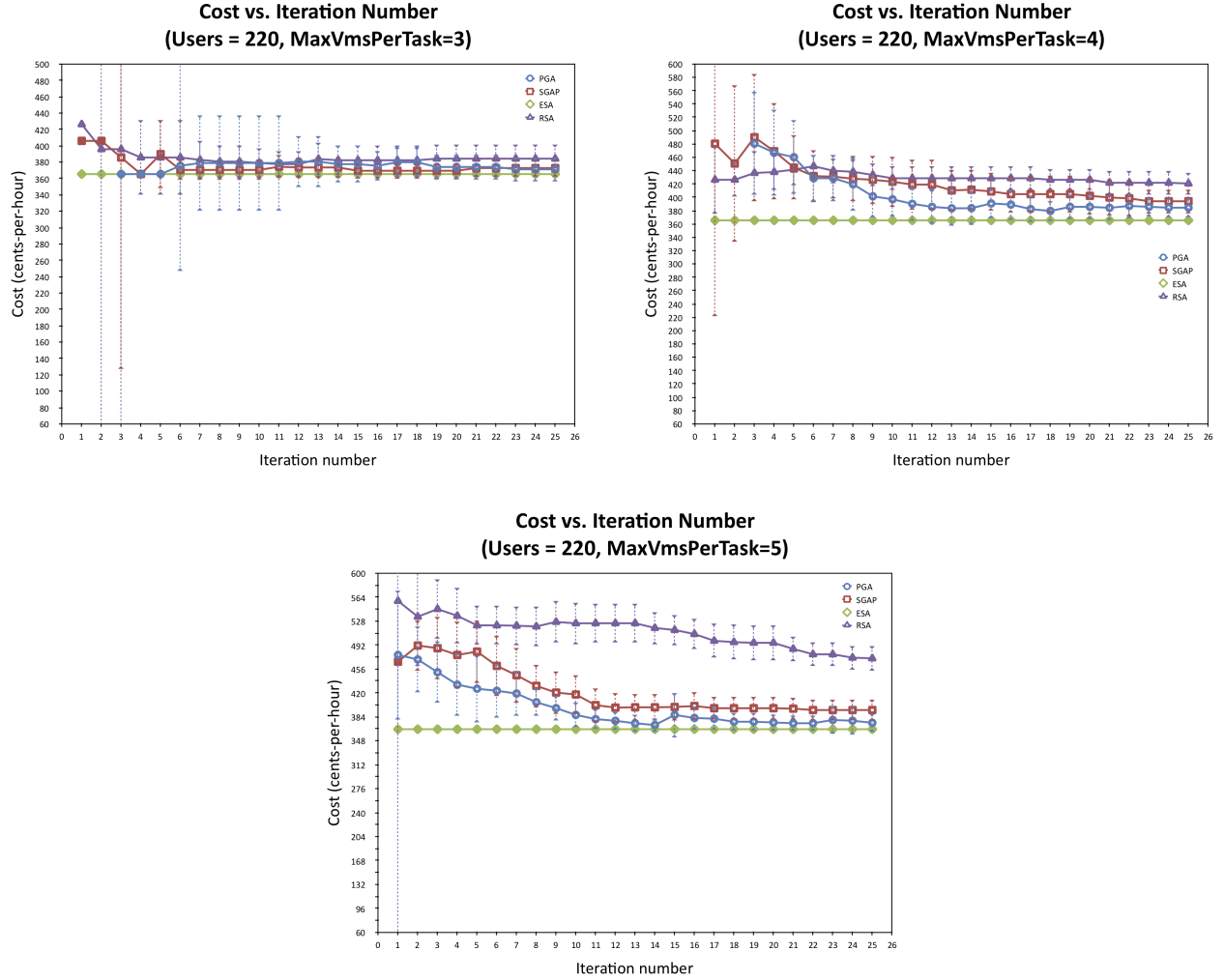


Figure 6.6: Cost vs. Iterations (Users=220)

possibility to find a feasible solution is higher. The number of evaluations performed by ESA in Table 6.6 shows the options in the search space.

With increase in search space, the %-error also increases. For  $MaxVmsPerTask = 2$ , the mean %-error for PGA, RSA and SGAP were: 2.0%, 5.0% and 4.3%, respectively. For  $MaxVmsPerTask = 5$ , the mean %-error were: 3.6%, 52.5% and 13.1%. As seen from the graphs, in comparison to other algorithms, RSA performs poorly as the search space increases. PGA shows the best results. This is also seen across the 23 experiments, the mean %-error for PGA, RSA and SGAP were: 2.6%, 27.3%, and 8.8%.

In each graph, with increasing users the cost of the VMs increase. This is directly related to the increase in demand that the application experiences, which causes VMs to be provisioned by the

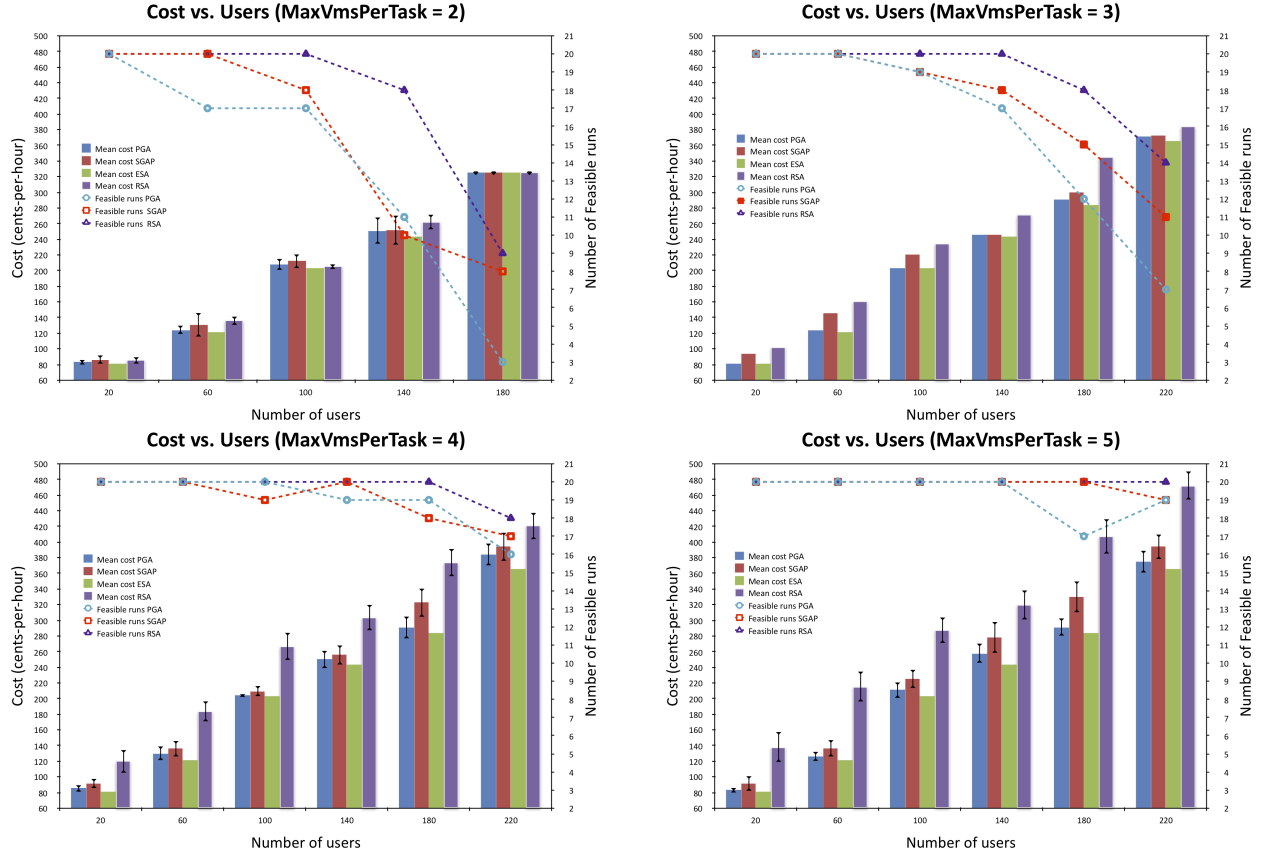


Figure 6.7: Cost vs. Users

decision maker.

It is also seen that within each graph, the number of feasible runs decrease within increase in users as  $MaxVmsPerTask$  is kept constant. This is because the search space remains constant here, but the feasible options decrease with higher workload.

ESA provides a yardstick of optimum cost. From the 23 experiments, PGA meets optimum mean cost 3 times, whereas SGAP and RSA meet the optimum once.

Similar results are depicted in graphs for Cost vs.  $MaxVmsPerTask$ . Figure 6.8 shows the graphs where  $MaxVmsPerTask$  are varied across each graph but User counts are kept constant. Appendix C shows results for user counts other than 20 and 220.

Overall, the %-error for PGA ranges between 0.0%–6.7%, RSA ranges between 0.0%–76.7%, and %-error for SGAP ranges between 0.0%–20.0%.

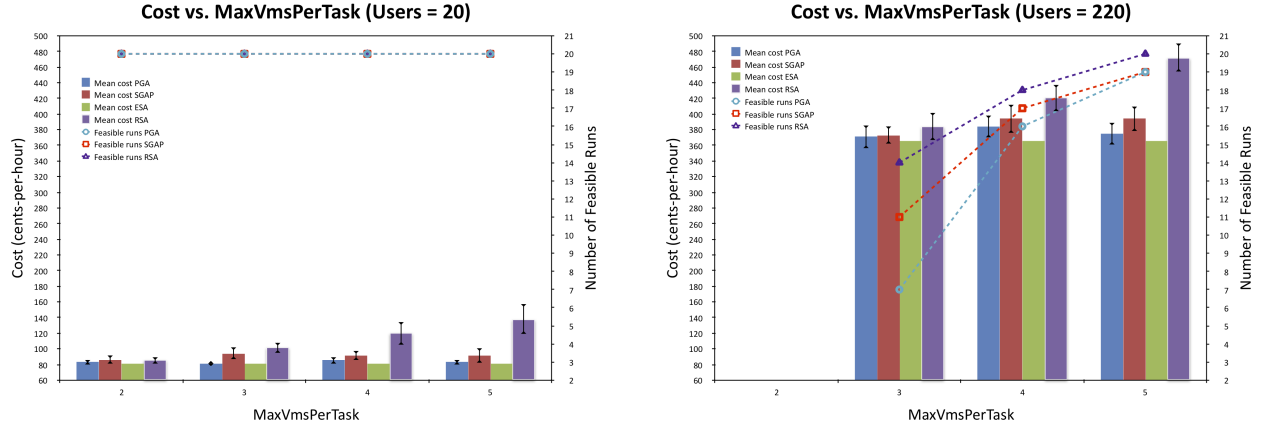


Figure 6.8: Cost vs. MaxVmsPerTask (Users=20 and 220)

### 6.3.5 Runtime vs Users

Figure 6.9, shows the Runtime vs. Users graphs. For PGA, RSA and SGAP, the reported runtime is for one run, which is found by dividing the total 20 runs runtime by 20.

ESA takes the longest time to find a solution. When users are equal to 20 and *MaxVmsPerTask* is 2, ESA completes in about 4 seconds, but when users are 220 and *MaxVmsPerTask* is 5, the algorithm runs for over 5 hours. This is directly related to the number of evaluations that ESA performs as the algorithm traverses through the complete search space. Table 6.6 shows the number of evaluations performed by PGA, RSA, SGAP and ESA. The equation for finding the number of evaluations by each algorithm is depicted in Section 5.3. Across graphs, increase in *MaxVmsPerTask* causes increase in runtime for all the algorithms.

MaxVmsPerTask	PGA	RSA	SGAP	ESA
2	500	500	500	728
3	500	500	500	4095
4	500	500	500	15624
5	500	500	500	46655

Table 6.6: Number of evaluations for the algorithms

From the graphs we see that for the same *MaxVmsPerTask*, the runtime also increases with increase in user counts. This is because the problem model solving time is seen to increase with higher user counts.

For 20 users and *MaxVmsPerTask* = 2, PGA takes under two seconds, whereas both SGAP

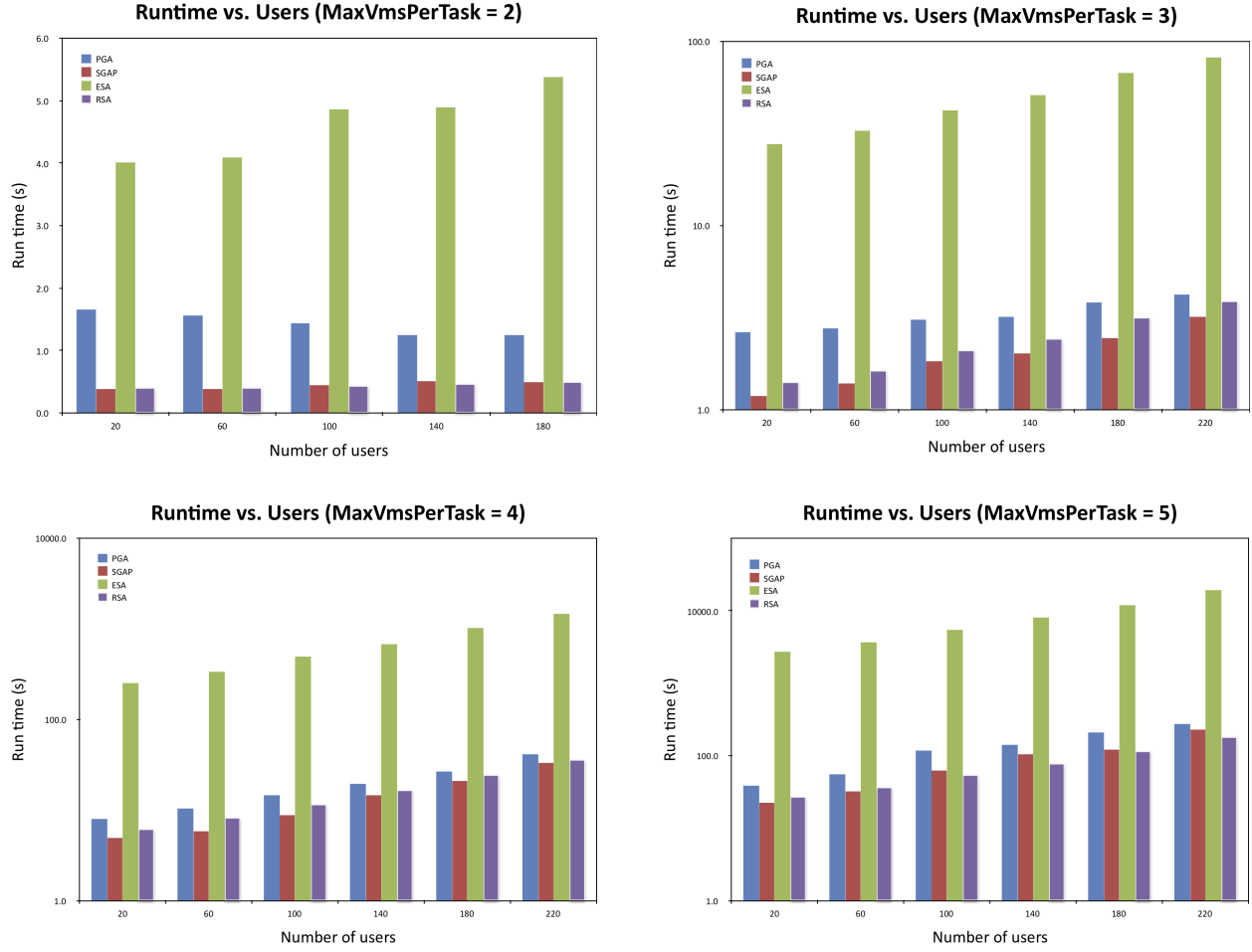


Figure 6.9: Runtime vs. Users. The runtime is of one complete run and not of 20 runs. Except for the first graph, Runtime is in logarithmic scale.

and RSA take less than half a second. For this case, ESA finds a solution in about 4 seconds.

For 220 users and  $MaxVmsPerTask = 5$ , PGA runs under 5 minutes per run, SGAP runs under 4 minutes and RSA runs under 3 minutes. For this case, ESA would run for over 5 hours to find the optimal solution. These results show that PGA runs longer than SGAP. LBM and LBX both make extra calls to the model solver and perform extra evaluations which explains the longer runtime for PGA. Since ESA is exhaustive search, the runtime increases exponentially.

## 6.4 Conclusions

A case study of a bike routes web application model using the decision maker has been presented in this chapter. The performance model from the application is derived first and fed into the decision

maker. Based on the user workloads, the decision maker decides a deployment configuration, consisting of task-to-VM allocations, to meet mean performance constraints and to minimize cost. The presented results show that in comparison to [RSA](#) and [SGAP](#), [PGA](#) shows the best performance by meeting performance constraints with a mean %-error of only 2.6%. Mean %-error of RSA is 27.3%, %-error of [SGAP](#) is 8.8%. SGAP, RSA and PGA run faster than ESA because of evaluating fewer configurations. PGA presents better results than SGAP at the cost of slightly slower runtimes.



# Chapter 7

## Conclusions

This thesis presents research on novel VM dynamic provisioning algorithms that provide an [AP](#) the means to automatically manage their application’s scalability to meet performance constraints and minimize cost when faced with changing workload. This chapter summarizes the work presented in this thesis.

**Chapter Outline** Section [7.1](#) lists the contributions made. Section [7.2](#) lists the limitations of the proposed algorithms. Section [7.3](#) suggests future work.

### 7.1 Contributions

The two main contributions of this thesis are summarized below.

#### 7.1.1 SatisfyQoS

In Chapter [4](#), the VM provisioning problem is formulated as a [CSP](#) to meet mean performance constraints. The presented solution, [SatisfyQoS](#) algorithm, uses layered bottlenecks to find bottlenecks and then perform horizontal or vertical scaling to solve the problem. [SatisfyQoS](#) algorithm demonstrates the strength of using layered bottlenecks in dynamic provisioning. In the case study that uses [SatisfyQoS](#), two cases are tried. Case 1 uses layered bottlenecks and Case 2 uses saturation values to determine the bottlenecks. It is found that although Case 1 requires more loops through the algorithm, the resulting model uses fewer resources while meeting the desired constraints.

### 7.1.2 PGA

In Chapter 5, the VM dynamic provisioning problem is formulated with focus on performance and cost. The goal of the problem is to minimize cost and meet the performance constraints. Four algorithms are presented as solutions which run within a decision maker component. PGA, is the main contribution where the other three algorithms, ESA, SGAP and RSA, are used for evaluating the performance of the former.

In Figure 5.4, a complete dynamic provisioning control system is presented. This picture shows all the components working together with the decision maker as the controller making the decisions. The workflow and the feedback cycle demonstrate a complete working system, which can be applied in practice. The workflow also shows the importance of each component, and with anyone missing the feedback cycle is broken.

The discussions that ensue in Section 5.4, elaborate further on the maintenance, troubleshooting and enhancements to such a system, allowing to reader to envision the reality of the system.

The PGA, is based on genetic algorithms and on the BStrength metric of layered bottlenecks. The use the BStrength metric differentiates PGA from SGAP. PGA uses the BStrength metric in two new operators: LBX and LBM, which perform crossover and mutation, respectively.

ESA is an exhaustive search and finds an optimal solution. RSA is a random population-based search.

In Chapter 6, a case study of a bike routes web application model is conducted, on which the four decision maker algorithms have been executed.

Across the 23 experiments, which have yielded feasible runs, the mean %-error for PGA, RSA and SGAP were: 2.6%, 27.3%, and 8.8%. Overall, the %-error for PGA ranges between 0.0%–6.7%, RSA ranges between 0.0%–76.7%, and %-error for SGAP ranges between 0.0%–20.0%.

For 20 users and  $MaxVmsPerTask = 2$ , PGA takes under two seconds, whereas both SGAP and RSA take less than half a second. For this case, ESA finds a solution in about 4 seconds.

For 220 users and  $MaxVmsPerTask = 5$ , PGA runs under 5 minutes per run, SGAP runs under 4 minutes and RSA runs under 3 minutes. For this case, ESA would run for over 5 hours to find the optimal solution. These results show that PGA runs longer than SGAP. LBM and LBX both make extra calls to the model solver and perform extra evaluations which explains the longer runtime for

PGA. Since ESA is exhaustive search, the runtime increases exponentially.

The case study demonstrates that PGA generates the best results in comparison to SGAP and RSA. In particular, PGA results in better cost than SGAP, at the expense of slightly longer runtime. All these algorithms run faster than the ESA and this behaviour is specially visible for larger user counts.

## 7.2 Limitations

[SatisfyQoS](#) was implemented to showcase the applicability of bottleneck theory in solving dynamic provisioning problems. The algorithm has a few limitations. It does not incorporate deallocation of VMs and processors. Furthermore, the algorithm also does not include cost in the decision making.

[PGA](#) solves a complete provisioning problem. The problem includes mean performance constraints and cost minimization as an objective while allowing for both allocation and deallocation of VMs. Section [5.4.3](#) presents the limitations of [PGA](#). Presently, this algorithm do not have a provision for accommodating flash crowds. Furthermore, using mean response time constraints does not work well for response times with long-tail distribution.

## 7.3 Future work

This section suggests the future work.

### 7.3.1 Multi-cloud

In Section [3.6](#), the thesis mentions about the recent interest in multi-clouds. Through the use of VM-types and their associated cost, our proposed solution can handle with a finer granularity a cost-effective configuration across different clouds. However, this decision would have been made without considering the network delays involved. The current problem only allows a static software architecture for the application but with multiple clouds, this architecture would change depending on which cloud's VM the application task is running on.

### **7.3.2 Minimal change**

Constantly changing the deployment configurations because of the provisioning decisions is an overhead. Spinning a new VM, running checks on it, stopping connections to the running task on the old VM and switching to use the new VM as part of the automatic blue-green deployment process takes a considerable amount of resources.

The proposed problem does not consider this overhead. Minimizing changes as part of the problem could help with easing this concern.

### **7.3.3 Adding Cost constraint**

The objective of the decision maker is to minimize cost, however, cost can also be added as a constraint. This is useful when the AP has a certain budget allocated for each application.

### **7.3.4 Discrete-Event-Simulation**


As a future work, a discrete-event-simulator, which simulates different workload patterns over time and records the response of the decision maker, would be quite useful for evaluating the decision maker algorithms. Other components could also be added to simulate a complete dynamic provisioning control loop system. This process may provide further insight into the decision making process and the system in general.

# Appendices


# Appendix A

## Permissions to use copyrighted material

### A.1 IEEE Permissions - UCC 2012 paper



HomeCreate AccountHelp



**Title:** Using Layered Bottlenecks for Virtual Machine Provisioning in the Clouds  
**Conference Proceedings:** 2012 IEEE Fifth International Conference on Utility and Cloud Computing  
**Author:** Yasir Shoaib  
**Publisher:** IEEE  
**Date:** Nov. 2012  
Copyright © 2012, IEEE


**LOGIN**  
If you're a copyright.com user, you can login to RightsLink using your copyright.com credentials. Already a RightsLink user or want to [learn more?](#)

BACK


CLOSE WINDOW

Figure A.1: IEEE Permissions for UCC 2012 paper

## A.2 ENTCS permission - 2011 paper



[Home](#)[Create Account](#)[Help](#)



**Title:** Web Application Performance Modeling Using Layered Queueing Networks

**Author:** Yasir Shoaib, Olivia Das

**Publication:** Electronic Notes in Theoretical Computer Science

**Publisher:** Elsevier

**Date:** 27 September 2011

Copyright © 2011 Elsevier B.V.

[LOGIN](#)

If you're a **copyright.com** user, you can login to RightsLink using your copyright.com credentials. Already a **RightsLink** user or want to [learn more?](#)

Please note that, as the author of this Elsevier article, you retain the right to include it in a thesis or dissertation, provided it is not published commercially. Permission is not required, but please ensure that you reference the journal as the original source. For more information on this and on your other retained rights, please visit: <https://www.elsevier.com/about/our-business/policies/copyright#Author-rights>

[BACK](#)[CLOSE WINDOW](#)

Copyright © 2018 [Copyright Clearance Center, Inc.](#) All Rights Reserved. [Privacy statement](#). [Terms and Conditions](#).  
Comments? We would like to hear from you. E-mail us at [customercare@copyright.com](mailto:customercare@copyright.com)

Figure A.2: ENTCS Permissions for 2011 paper

# Appendix B

## Software architecture: The Decision Maker

This appendix discusses the software architecture of the decision maker.

**Appendix Outline** Section [B.1](#) explains the specific requirements that have driven the choice of components and design decisions. Section [B.2](#) mentions the component dependencies of the decision maker. Section [B.3](#), Section [B.4](#), and Section [B.5](#) explains the design and implementation of jLQNInterface library, application-cloud-model library and the decision maker. Section [B.6](#) mentions about the MOEA framework. Section [B.7](#) are the conclusions.

### B.1 Requirements and Design decisions

The role of the decision maker within the dynamic provisioning control system has been explained in the previous section. In this section we discuss the requirements and the design decisions governing the design of the decision maker.

For decision making, genetic algorithms have been chosen. Based on the problem we are solving, the algorithm requires knowing the following to reach a decision:

- application architecture
- container types that are available from the cloud
- constraints and objectives

The output decision is conveyed in the form of a deployment configuration.

To represent the application architecture and the deployment configuration, our choice is to use LQN models. Since, we will be implementing the decision maker as a software, we need APIs for building, solving, analyzing and manipulating LQN models from within the software program.



[jLQNInterface](#) library is a tool which we have developed for this purpose. The features have been discussed in Chapter 2 and Chapter 4.

[jLQNInterface](#) is a Java library, therefore using the library means that Java becomes the default choice to implement the decision maker.

For the genetic algorithm we have chosen the [MOEA](#) framework. The constraints and objectives for the algorithm will be specified through the decision maker application. As the genetic algorithm is run through the [MOEA](#) framework, it calls implementations within decision maker for completion of the algorithm.

Using LQN models directly has some challenges however. LQN models allow for fine grained description of the application architecture through processors, tasks, entries and activities. But, there are no high-level abstractions available to represent container types and clouds. A model interface that allows for a simplified representation of applications, container types and clouds is necessary to ease the input generation process. For this purpose, we have developed the [application-cloud-model](#) library.

Table B.1 lists the requirements that need to be fulfilled for developing the decision maker and the corresponding components that implement those requirements.

Table B.1: Decision maker requirements and the corresponding components that fulfil those requirements

Requirement	Component
representation of the application architecture input	LQN models
tool that allows solving, analyzing and manipulating LQN models automatically	<a href="#">jLQNInterface</a> library
model interface that allows for simplified representation of applications, container types and clouds	<a href="#">application-cloud-model</a> library
genetic algorithm framework	<a href="#">MOEA</a> framework
constraints and objectives for the algorithm	<a href="#">decision maker</a>

## B.2 Software components

In the previous section, we listed different components that fulfil the requirements for building the decision maker. Figure B.1 shows how these components are related. The decision maker depends directly on the application-cloud-model and the MOEA Framework. The application-cloud-model depends on the jLQNInterface. And, the jLQNInterface depends on LQNS. All the projects except

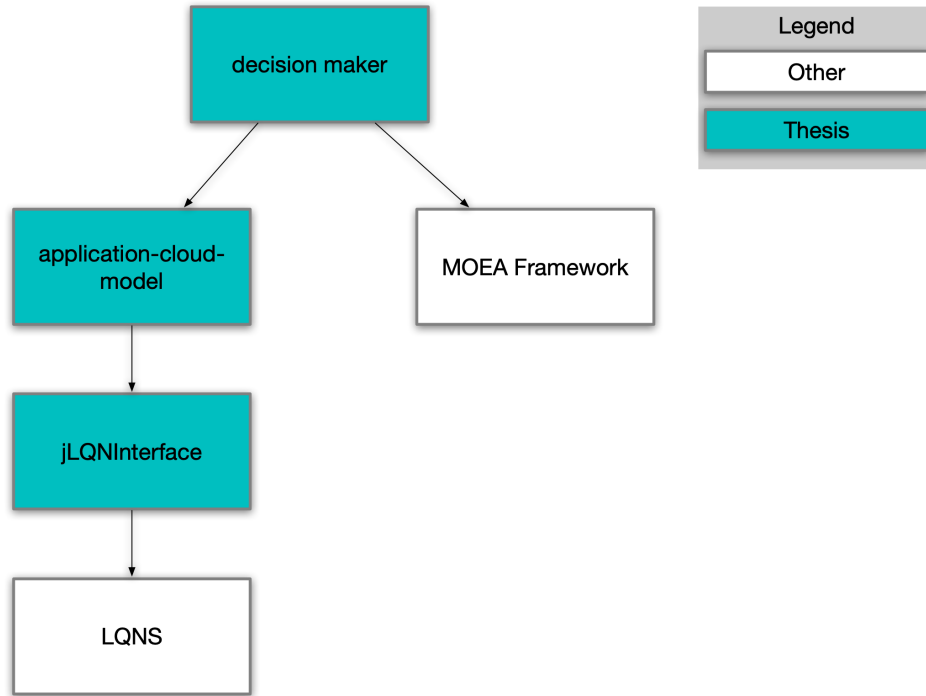


Figure B.1: Decision maker dependency diagram: This thesis explains three of the components which we have developed: decision maker, application-cloud-model and the jLQNInterface. MOEA Framework and LQNS are externally developed; these are direct and transitive dependencies of the decision maker, respectively

LQNS tool are [maven projects](#) and expose their functionalities through their public APIs. LQNS runs as a command-line tool and is invoked in jLQNInterface as a shell command. Table [B.2](#) lists details about each component.

### B.2.1 Maven projects

As part of the release process, [maven projects](#) are packaged as pom/jar files and published to the Central repository (<https://search.maven.org>). These published artifacts can then easily be pulled in as dependencies of other maven projects. For example, adding following lines to the dependencies section of the [pom](#) of a maven project will include application-cloud-model as a dependency.

Listing B.1: Adding application-cloud-model version 3.0.4 as a dependency

```

1 <dependency>
2   <groupId>ca.appsimulations</groupId>
3   <artifactId>models</artifactId>
4   <version>3.0.4</version>
5 </dependency>

```

Table B.2: Software Components

Component	Source	License	Author(s)	Packaging
LQNS	<a href="https://github.com/layeredqueuing/V5">https://github.com/layeredqueuing/V5</a>  249,617 lines-of-code (Prolog, C, C++)  version 5.10	<a href="https://github.com/layeredqueuing/V5/blob/master/dist/license.txt">https://github.com/layeredqueuing/V5/blob/master/dist/license.txt</a>	Franks et al. <a href="https://github.com/layeredqueuing/V5/blob/master/AUTHORS">https://github.com/layeredqueuing/V5/blob/master/AUTHORS</a>	executable
jLQNInterface	<a href="https://github.com/yshoaib/jLQNInterface">https://github.com/yshoaib/jLQNInterface</a>  4,513 lines-of-code (Java)  version 1.1.0	Apache License 2.0	Yasir Shoaib	pom,jar
application-cloud-model	<a href="https://github.com/yshoaib/application-cloud-model">https://github.com/yshoaib/application-cloud-model</a>  1,668 lines-of-code (Java)  version 3.0.4	Apache License 2.0	Yasir Shoaib	pom,jar
MOEA Framework	<a href="https://github.com/MOEAFramework/MOEAFramework">https://github.com/MOEAFramework/MOEAFramework</a>  58,731 lines-of-code (Java)  version 2.12	LGPL v3.0 or later	David Hadka	pom,jar
decision maker	Private repository  1,332 lines-of-code (Java)  version 1.0.0-SNAPSHOT	Proprietary	Yasir Shoaib	pom,jar

As shown in Listing B.1, it is quite easy to add dependencies to a project. Each artifact that is hosted on the Central repository is versioned, therefore including a dependency requires stating the exact version of the artifact that is to be pulled in.

### B.2.2 Semantic versioning

All the three projects that have been implemented are semantically versioned. Following is a how semantic versioning works:

Consider a version format of X.Y.Z (Major.Minor.Patch). Bug fixes not affecting the API increment the patch version, backwards compatible API additions/changes increment the minor version, and backwards incompatible API changes increment the major version [139].

Our projects adhere to semantic versioning to help identify the kind of change across versions and to help the consumers of these components identify which versions are safe to upgrade to without breaking their software.

### B.2.3 Testing

Each component developed has test cases included within them that help with keeping changes in check. If a code change breaks an existing functionality which had a test case associated to it then that change will cause the test to fail. This will indicate that either the change is not good and needs fixing or its a breaking change that will require a new major version increment.

### B.2.4 Version control

In this thesis, all the software projects developed and dependencies are version controlled through [git](#). The decision maker is a private repository and hosted at <http://gitlab.com>. The other projects are hosted at <http://github.com>. The links for open-source software are presented in Table B.2.

## B.3 jLQNInterface library

jLQNInterface library was introduced in Section 4.2.2 and discussed about in Section B.1. The library provides direct access to LQN modeling and the LQNS solver. Pulling the library in as a dependency within a maven-based Java software, provides the quickest way to access the jLQNInterface

API. Listing B.2 shows the jLQNInterface pom. At the time of writing, version 1.1.0 of jLQNInterface has been published on Central repository for public access.

Listing B.2: jLQNInterface POM

```
1 <!-- https://mvnrepository.com/artifact/ca.appsimulations/jLQNInterface -->
2 <dependency>
3   <groupId>ca.appsimulations</groupId>
4   <artifactId>jLQNInterface</artifactId>
5   <version>1.1.0</version>
6 </dependency>
```

### B.3.1 Example

Once jLQNInterface is added to a project, a complete software that leverages performance models can be built. A simple example code is presented in Listing B.3. The program reads an LQN XML model, and for that model outputs the response time for different user counts. For completeness, the XML model is presented in Listing B.4 and shown pictorially in Figure B.2. jLQNInterface examples are available publicly at <https://github.com/yshoaib/jLQNInterfaceExamples>

Listing B.3: jLQNInterface Example

```
1 public static void main(String[] args) throws Exception {
2
3     String inputFile = "input.lqnx";
4     File intermediateInputFile = new File("intermediateInputFile.lqnx");
5     File outputFile = new File("output.lqxo");
6     File outputPs = new File("output.ps");
7     outputFile.delete();
8     outputPs.delete();
9
10    int i =1;
11    int users = 1;
12    do{
13
14        //read model
15        LqnModel lqnModel = new LqnModel();
16        new LqnInputParser(lqnModel,
17            true).parseFile(getResourceFile(inputFile).getAbsolutePath());
18
19        //set user count
20        lqnModel.entryByName("load").getTask().setMultiplicity(users);
21        log.info("Users: " +
22            lqnModel.entryByName("load").getTask().getMutiplicityString());
```

```

21
22
23 //write intermediate input
24 new LqnModelWriter().write(lqnModel, intermediateInputFile.getAbsolutePath());
25
26
27 //solve
28 boolean solveResult =
29     LqnSolver.solveLqns(intermediateInputFile.getAbsolutePath(),
30                         new LqnResultParser(new LqnModel()),
31                         outputFile.getAbsolutePath());
32
33 //was the model solved successfully
34 if (solveResult == false) {
35     log.error("problem solving lqn model");
36     return;
37 }
38
39 //find response time
40 LqnModel lqnModelResult = new LqnModel();
41 new LqnResultParser(lqnModelResult).parseFile(outputFile.getAbsolutePath());
42 log.info("response time of load is: " + getResponseTime(lqnModelResult,
43     "load"));
44
45 LqnSolver.savePostScript(outputFile.getAbsolutePath(),
46     outputPs.getAbsolutePath());
47
48 users = i * 10; //users = 1, 10, 20, ..., 100
49 i++;
50 }while(i <= 10);
51 }
52
53 public static double getResponseTime( LqnModel lqnModelResult, String entryName) {
54     String activityName =
55         lqnModelResult.entryByName(entryName).getEntryPhaseActivities().getActivityAtPhase(1)
56         .getName();
57     List<ActivityDefBase> activities =
58         lqnModelResult.activities().stream().filter(activityDefBase ->
59             activityDefBase.getName().equals
60             (activityName)).collect(
61             toList());
62
63     double responseTime = 0;
64     if (activities.size() > 0) {
65         ActivityPhases ap = (ActivityPhases) activities.get(0);
66         responseTime = ap.getResult().getService_time();
67     }
68     return responseTime;
69 }

```

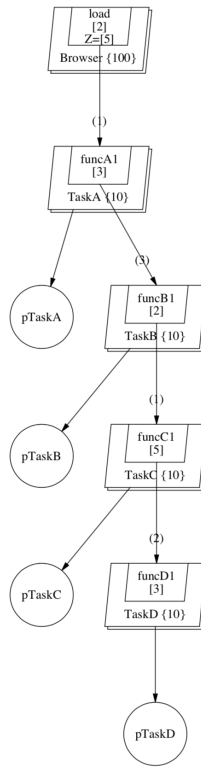


Figure B.2: Simple LQN model

Listing B.4: Simple LQN model

```

1 <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2 <lqn-model xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" name="input-rep"
3   description="simple model"
4   xsi:noNamespaceSchemaLocation="lqn.xsd">
5   <solver-params comment="comment" conv_val="0.01" it_limit="50000"
6     underrelax_coeff="0.9" print_int="1"/>
7   <processor name="pClient" scheduling="inf">
8     <task name="Browser" scheduling="ref" multiplicity="100">
9       <entry name="load" type="PH1PH2">
10         <entry-phase-activities>
11           <activity name="load_1" phase="1" host-demand-mean="2.0"
12             think-time="5.0">
13             <synch-call dest="funcA1" calls-mean="1.0"/>
14           </activity>
15         </entry-phase-activities>
16       </entry>
17     </task>
18   </processor>
19   <processor name="pTaskA" scheduling="ps" quantum="0.2">
20     <task name="TaskA" scheduling="fcfs" multiplicity="10">
21       <entry name="funcA1" type="PH1PH2">
22         <entry-phase-activities>
23           <activity name="funcA1_1" phase="1" host-demand-mean="3.0">

```

```

24         </activity>
25     </entry-phase-activities>
26 </entry>
27 </task>
28 </processor>
29 <processor name="pTaskB" scheduling="ps" quantum="0.2">
30     <task name="TaskB" scheduling="fcfs" multiplicity="10">
31         <entry name="funcB1" type="PH1PH2">
32             <entry-phase-activities>
33                 <activity name="funcB1_1" phase="1" host-demand-mean="2.0">
34                     <synch-call dest="funcC1" calls-mean="1.0"/>
35                 </activity>
36             </entry-phase-activities>
37         </entry>
38     </task>
39 </processor>
40 <processor name="pTaskC" scheduling="ps" quantum="0.2">
41     <task name="TaskC" scheduling="fcfs" multiplicity="10">
42         <entry name="funcC1" type="PH1PH2">
43             <entry-phase-activities>
44                 <activity name="funcC1_1" phase="1" host-demand-mean="5.0">
45                     <synch-call dest="funcD1" calls-mean="2.0"/>
46                 </activity>
47             </entry-phase-activities>
48         </entry>
49     </task>
50 </processor>
51 <processor name="pTaskD" scheduling="ps" quantum="0.2">
52     <task name="TaskD" scheduling="fcfs" multiplicity="10">
53         <entry name="funcD1" type="PH1PH2">
54             <entry-phase-activities>
55                 <activity name="funcD1_1" phase="1" host-demand-mean="3.0"/>
56             </entry-phase-activities>
57         </entry>
58     </task>
59 </processor>
60 </lqn-model>

```

### B.3.2 Classes

With jLQNInterface all interactions with the LQN models that reside on disk happen through their XML representation (a.k.a. LQNX model). Plain LQN models can be readily converted to LQNX model through the LQNS tool. The LQN XML elements and attributes are mapped as classes and enumeration within jLQNInterface. Along with the elements and attributes, the library includes other classes that work with the LQN model. The example code in Listing B.4 uses the following classes provided by the jLQNInterface library:



**LqnModel** class represents LQN models, comprising of processors, tasks, entries, etc.

**LqnModelWriter** writes the in-memory Java objects to LQN model files on disk

**LqnSolver** solves the model

**LqnInputParser** parses the LQN model on disk into a Java object in-memory

**LqnResultParser** parses the solved LQN model results on disk into a Java object in-memory

## B.4 application-cloud-model library

Section B.1 discussed about the need for application-cloud-model library. To build and interact with LQN models, the application-cloud-model library pulls in jLQNInterface library as a dependency. Listing B.5 shows the application-cloud-model [pom](#), which could be added to a software to pull in this library as a dependency. At the time of writing, version 3.0.4 of application-cloud-model has been published on Central repository for public access.

Listing B.5: application-cloud-model POM

```
1 <!-- https://mvnrepository.com/artifact/ca.appsimulations/models -->
2 <dependency>
3   <groupId>ca.appsimulations</groupId>
4   <artifactId>models</artifactId>
5   <version>3.0.4</version>
6 </dependency>
```

### B.4.1 Example

Just like jLQNInterface, the application-cloud-model library can be used to build a complete software based on performance models. Since application-cloud model library includes jLQNInterface as a dependency, any software that pulls in application-cloud-model also has access to jLQNInterface library.

Listing B.6 shows the code for building an application. Listing B.7 lists the code for building a cloud with container images and containers. Within the library, containers and containerImages run within the cloud, however, these could also be considered as VMs and VM-images for our

purposes in this thesis. Listing B.8, shows the main code that uses both the built application and the built cloud to evaluate the model for different user counts. The example here presents the same model shown earlier in Figure B.2. The library examples are available publicly at <https://github.com/yshoaib/modelsExamples>

Listing B.6: application-cloud-model example (buildApp)

```
1 public static App buildApp(String appName, int users, int maxReplicas, double
   responseTimeObjective){
2     App app = AppBuilder.builder()
3         .name(appName)
4         .maxReplicas(maxReplicas)
5         .responseTimeObjective(responseTimeObjective)
6         .service("Browser", users)
7         .serviceEntry("load", "load_1", 2.0, 7.0)
8         .buildService()
9         .service("TaskA", 10)
10        .serviceEntry("funcA1", "funcA1_1", 3.0)
11        .buildService()
12        .service("TaskB", 10)
13        .serviceEntry("funcB1", "funcB1_1", 2.0)
14        .buildService()
15        .service("TaskC", 10)
16        .serviceEntry("funcC1", "funcC1_1", 5.0)
17        .buildService()
18        .service("TaskD", 10)
19        .serviceEntry("funcD1", "funcD1_1", 3.0)
20        .buildService()
21        .call("call1", "Browser", "TaskA", "load", "funcA1", 1)
22        .call("call2", "TaskA", "TaskB", "funcA1", "funcB1", 3)
23        .call("call3", "TaskB", "TaskC", "funcB1", "funcC1", 1)
24        .call("call4", "TaskC", "TaskD", "funcC1", "funcD1", 2)
25        .build();
26
27     return app;
28 }
```

Listing B.7: application-cloud-model example (buildCloud)

```
1 public static Cloud buildCloud(App app){
2     Cloud cloud = CloudBuilder.builder()
3         .name("cloud1")
4         .containerTypes(asList(small, medium, large))
5         .containerImage("Browser")
6         .service("Browser", app)
7         .buildContainerImage()
8         .containerImage("TaskA")
```

```

9      .service("TaskA", app)
10     .buildContainerImage()
11     .containerImage("TaskB")
12     .service("TaskB", app)
13     .buildContainerImage()
14     .containerImage("TaskC")
15     .service("TaskC", app)
16     .buildContainerImage()
17     .containerImage("TaskD")
18     .service("TaskD", app)
19     .buildContainerImage()
20     .build();
21
22     cloud.instantiateContainer("pClient", "Browser", small);
23     cloud.instantiateContainer("pTaskA", "TaskA", small);
24     cloud.instantiateContainer("pTaskB", "TaskB", small);
25     cloud.instantiateContainer("pTaskC", "TaskC", small);
26     cloud.instantiateContainer("pTaskD", "TaskD", small);
27     return cloud;
28 }

```

Listing B.8: application-cloud-model example

```

1     public static void main(String[] args) throws Exception {
2
3         File inputFile = new File("input.lqnx");
4         LqnXmlDetails xmlDetails = buildLqnXmlDetails();
5         SolverParams solverParams = buildSolverParams();
6
7         File intermediateInputFile = new File("intermediateInputFile.lqnx");
8         File outputFile = new File("output.lqxo");
9         File outputPs = new File("output.ps");
10        outputFile.delete();
11        outputPs.delete();
12
13        int i =1;
14        int users = 1;
15        do{
16            App testApp = buildApp("testApp", users, 5, 50000.0);
17            Cloud testCloud = buildCloud(testApp);
18
19            LqnModel lqnModel = LqnModelFactory.build(testApp, xmlDetails, solverParams);
20            new LqnModelWriter().write(lqnModel, inputFile.getAbsolutePath());
21
22            log.info("Users: " +
23                    lqnModel.entryByName("load_1").getTask().getMutiplicityString());
24
25            //write intermediate input
26            new LqnModelWriter().write(lqnModel, intermediateInputFile.getAbsolutePath());
27
28            //solve

```

```

29         boolean solveResult =
30             LqnSolver.solveLqns(intermediateInputFile.getAbsolutePath(),
31                                 new LqnResultParser(new LqnModel()),
32                                 outputFile.getAbsolutePath());
33
34         //was the model solved successfully
35         if (solveResult == false) {
36             log.error("problem solving lqn model");
37             return;
38         }
39
40         //find response time
41         LqnModel lqnModelResult = new LqnModel();
42         new LqnResultParser(lqnModelResult).parseFile(outputFile.getAbsolutePath());
43         log.info("response time of load_1 is: " + getResponseTime(lqnModelResult,
44                             "load_1"));
45
46         LqnSolver.savePostScript(outputFile.getAbsolutePath(),
47                                 outputPs.getAbsolutePath());
48
49         users = i * 10; //users = 1, 10, 20, ..., 100
50         i++;
51     }while(i <= 10);
52 }
53
54
55 private static SolverParams buildSolverParams() {
56     return SolverParams
57         .builder()
58         .comment(COMMENT)
59         .convergence(CONVERGENCE)
60         .iterationLimit(ITERATION_LIMIT)
61         .underRelaxCoeff(UNDER_RELAX_COEFF)
62         .printInterval(PRINT_INTERVAL)
63         .build();
64 }
65
66 private static LqnXmlDetails buildLqnXmlDetails() {
67     return LqnXmlDetails
68         .builder()
69         .name(XML_NAME)
70         .xmlnsXsi(XML_NS_URL)
71         .description(XML_DESCRIPTION)
72         .schemaLocation(SCHEMA_LOCATION)
73         .build();
74 }
75
76
77 public static double getResponseTime( LqnModel lqnModelResult, String entryName) {
78     String activityName =
79         lqnModelResult.entryByName(entryName).getEntryPhaseActivities().getActivityAtPhase(1)
80         .getName();
81     List<ActivityDefBase> activities =

```

```

81         lqnModelResult.activities().stream().filter(activityDefBase ->
82             activityDefBase.getName().equals
83             (activityName)).collect(
84                 toList());
85
86     double responseTime = 0;
87     if (activities.size() > 0) {
88         ActivityPhases ap = (ActivityPhases) activities.get(0);
89         responseTime = ap.getResult().getService_time();
90     }
91     return responseTime;
92 }

```

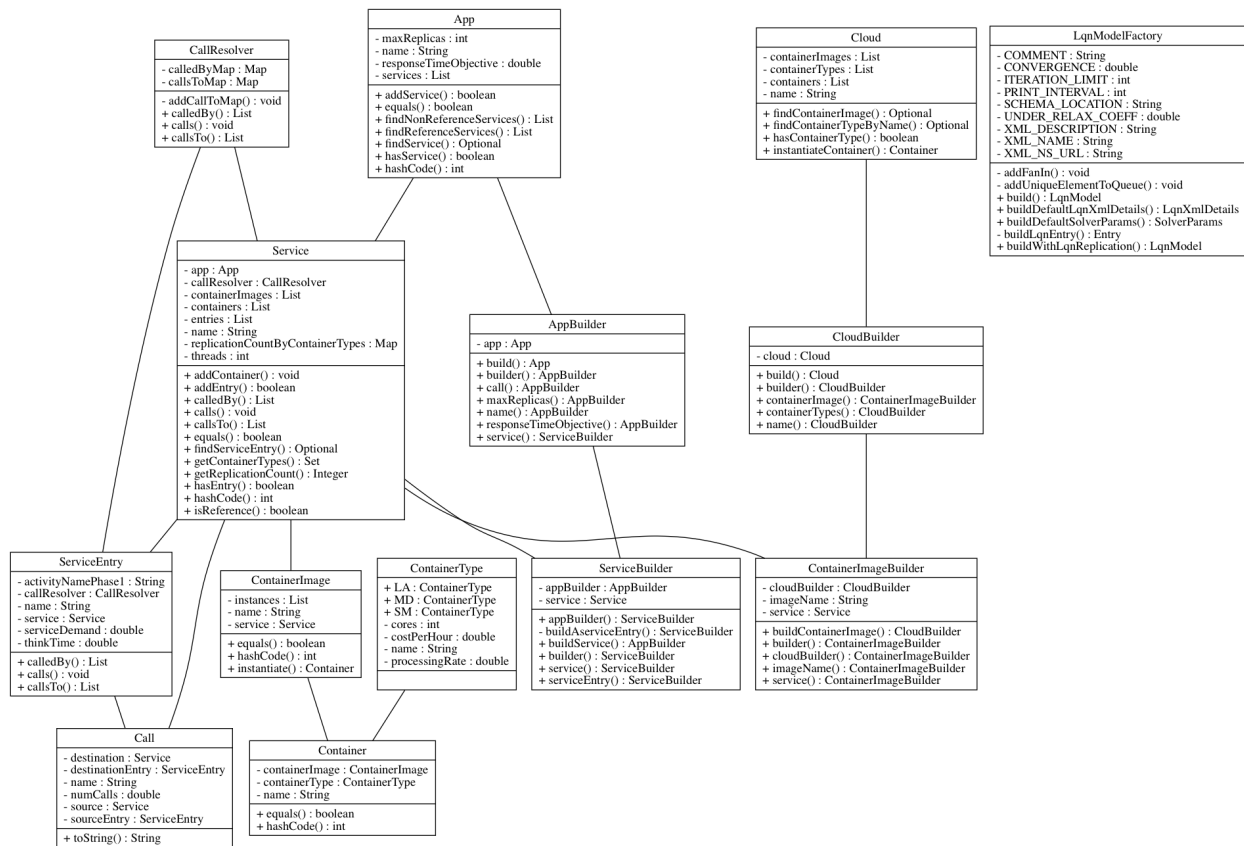


Figure B.3: application-cloud-model class diagram

## B.4.2 Class diagram

Figure B.3 shows the class diagram of the application-cloud-model library. The main classes are:

**App** application which has services

**Cloud** a cloud which has containers and container images.

**Service** represents a task in the application

**ServiceEntry** represents an entry of the service (task).

**Container** a running container or a VM

**ContainerImage** container image or a VM image

**ContainerType** represents a container-type or a VM-type.

**Call** represents calls from one service entry to another

**LqnModelFactory** used to convert the application-cloud-model into an LQN performance model

## B.5 The decision maker software

This section discusses the inputs and class diagram of the decision maker.

### B.5.1 Inputs

The decision maker requires various inputs for decision making. Listing B.9, the file that a configuration file that the AP would provide as input to the decision maker.

Listing B.9: Parameters

```
1 # Response time objective in seconds
2 responseTimeObjective=8.0
3 # Workload={number-of-users of class type 1, ...}
4 W={20}
5 # Maximum VMs per task
6 MaxVmsPerTask=2
7 # Algorithm type options: {"PGA", "SGAP", "RSA", "ESA"}
8 algorithmType=PGA
9 # Population size: vary from 10-60 (interval of 5), default value is: 20
10 P=20
11 # Iterations: vary from 5-25 (interval of 5); default value is: 20
12 I=20
13 # Mutation Distribution Index: vary from 5-25; default value is: 10.0
14 MDI=10.0
15 # Cross-over Distribution Index: vary from 5-25; default value is: 15.0
16 CDI=15.0
17 # Number of runs of the algorithm to derive average from
18 numberOfRuns=20
19 # VM-types={"name", num-processor, cost-rate-per-hour), ... }
```

20 VMTypes= {"small", 2, 20.3}, {"medium", 4, 40.6}, {"large", 8, 81.2}}

## B.5.2 Class diagram

Figure B.4 shows the class diagram of the decision maker. The main classes are:

**ProblemModel** evaluates based on the set objectives and generates a new deployment configuration.

**Variables** variables of the problem

**SolutionFactory** helper of the ProblemModel class for generating new deployment configurations

**ModifiedCrossover** LBX logic

**ModifiedMutation** LBM logic

**MainAppService** driver of the decision maker logic

Some of the classes are inherited from the MOEA framework. Also, the reason we do not directly see genetic algorithm classes is that this process is handled and run through the MOEA framework. Furthermore, those genetic algorithm classes are not shown here.

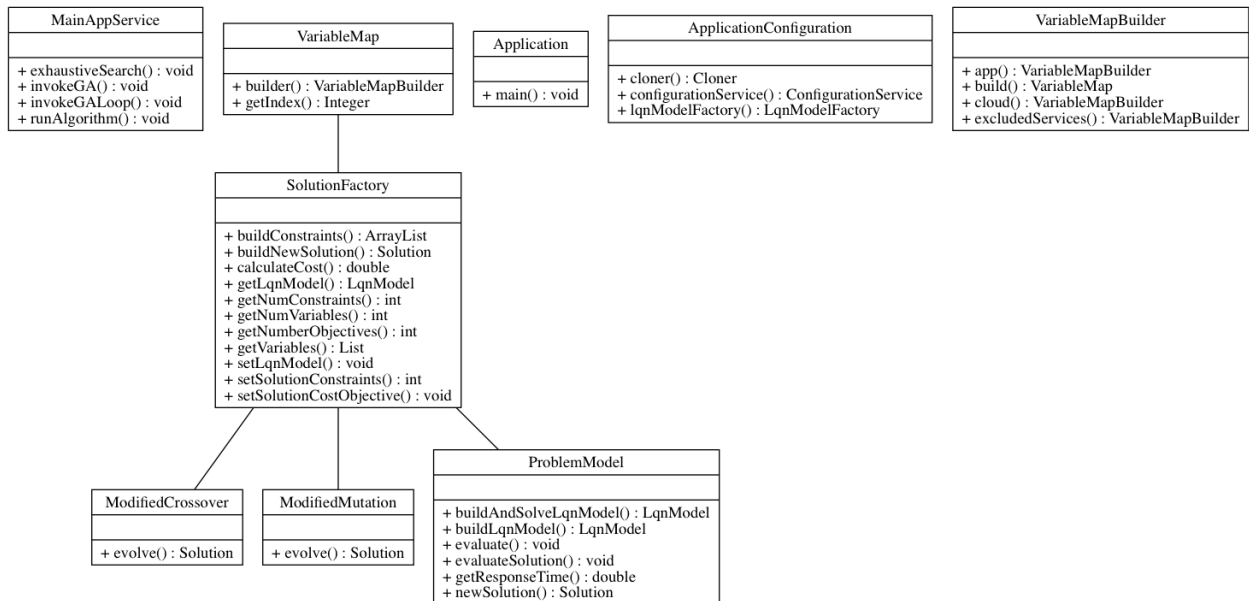


Figure B.4: decision-maker class diagram

## **B.6 MOEA Framework**

Implemented by David Hadka et al., MOEA Framework is an open-source Java library that allows implementing multi-objective evolutionary algorithms [[129](#), [140](#)].

## **B.7 Conclusions**

This appendix explains the software architecture of the decision maker and the libraries that we have developed. The requirements, the decision decisions and class diagrams have been explained to further help with understanding the design.



# Appendix C

## Results

This appendix presents raw data and extra graphs from the case study presented in this thesis.

### C.1 Data and graphs

MaxVmReplicas	Users	PGA	SGAP	RSA	ESA
		Run Time (s)	Run Time (s)	Run Time (s)	Run Time (s)
2	20	1.7	0.4	0.4	4
	60	1.6	0.4	0.4	4.1
	100	1.4	0.4	0.4	4.9
	140	1.2	0.5	0.5	4.9
	180	1.3	0.5	0.5	5.4
	220	infeasible	infeasible	infeasible	infeasible
3	20	2.6	1.2	1.4	27.7
	60	2.8	1.4	1.6	32.8
	100	3.1	1.8	2.1	42
	140	3.2	2	2.4	51.1
	180	3.8	2.5	3.1	67.4
	220	4.2	3.2	3.9	82.1
4	20	8.1	4.9	6.2	251.2
	60	10.4	5.9	8.2	339.7
	100	14.8	8.9	11.7	498.5
	140	19.4	14.8	16.5	683.5
	180	26.5	21.3	24.5	1012.4
	220	40.9	33.1	35.8	1469.8
5	20	38.7	22.5	27.3	2731.8
	60	55	32.1	37.1	3726.7
	100	119.9	62.5	53.6	5488.5
	140	143.3	105.9	77.3	7947.5
	180	212	121.4	114	11835.1
	220	276.8	230.3	179.2	19093.5

Table C.1: Runtime vs. Users

MaxVmReplicas	Users	PGA		SGAP		RSA		ESA
		Mean	Feasible	Mean	Feasible	Mean	Feasible	Mean
		Cost	Runs	Cost	Runs	Cost	Runs	Cost
2	20	83.23	20	85.26	20	86.28	20	81.2
	60	124.19	17	136.01	20	130.94	20	121.8
	100	207.78	17	205.03	20	212.02	18	203
	140	250.98	11	261.64	18	251.72	10	243.6
	180	324.8	3	324.8	9	324.8	8	324.8
	220	N/A	0	N/A	0	N/A	N/A	N/A
3	20	81.2	20	101.5	20	94.4	20	81.2
	60	123.83	20	160.37	20	146.16	20	121.8
	100	203	19	234.47	20	220.09	19	203
	140	245.99	17	271.01	20	245.86	18	243.6
	180	290.97	12	345.1	18	300.44	15	284.2
	220	371.2	7	384.25	14	372.78	11	365.4
4	20	85.26	20	119.77	20	91.35	20	81.2
	60	129.92	20	183.72	20	136.01	20	121.8
	100	204.01	20	266.94	20	209.41	19	203
	140	250.01	19	303.49	20	255.78	20	243.6
	180	290.61	19	373.52	20	322.54	18	284.2
	220	384.43	16	420.66	18	394.06	17	365.4
5	20	83.23	20	138.04	20	91.35	20	81.2
	60	125.86	20	215.18	20	136.01	20	121.8
	100	211.12	20	287.25	20	225.33	20	203
	140	257.81	20	319.73	20	278.11	20	243.6
	180	291.36	17	407.02	20	329.88	20	284.2
	220	375.02	19	471.98	20	394.25	19	365.4

Table C.2: Mean Cost vs. Users

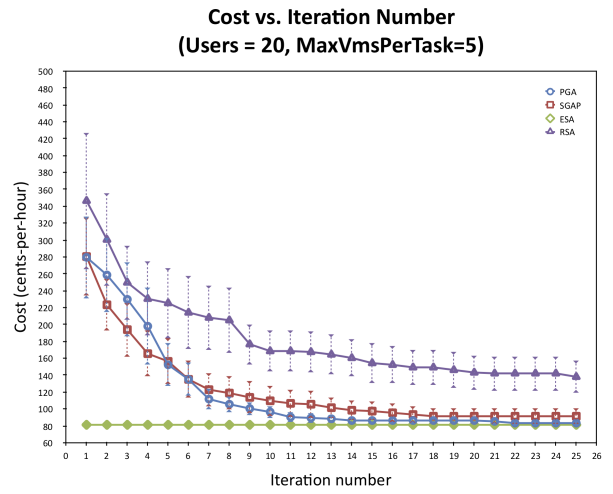
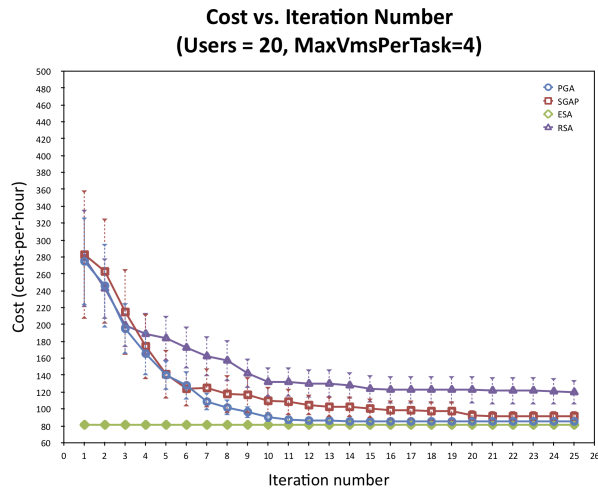
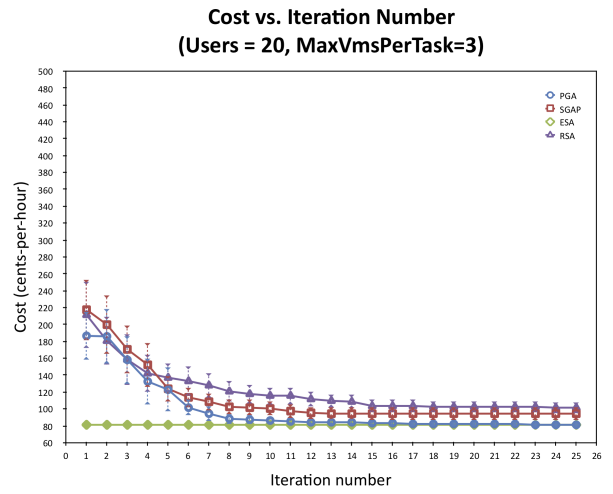
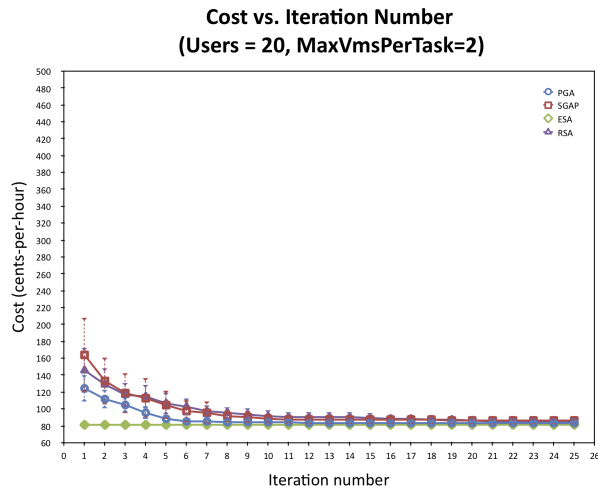


Figure C.1: Cost vs. Iterations (Users=20)

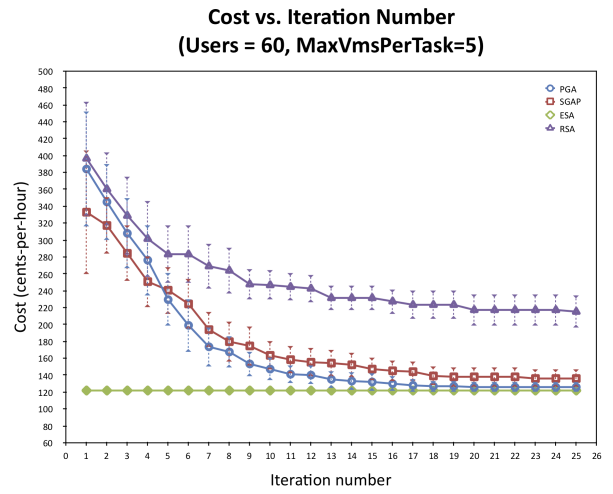
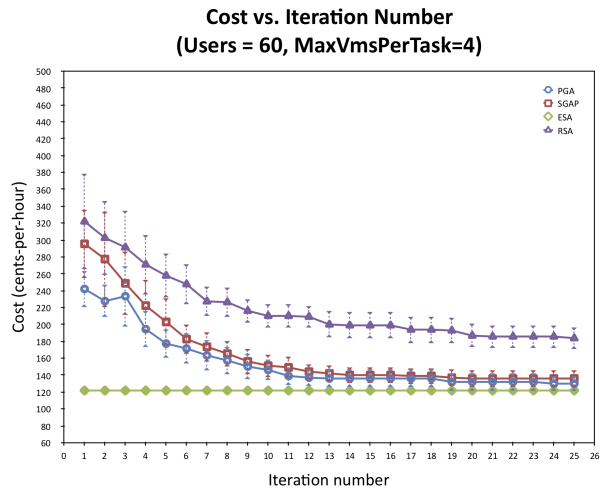
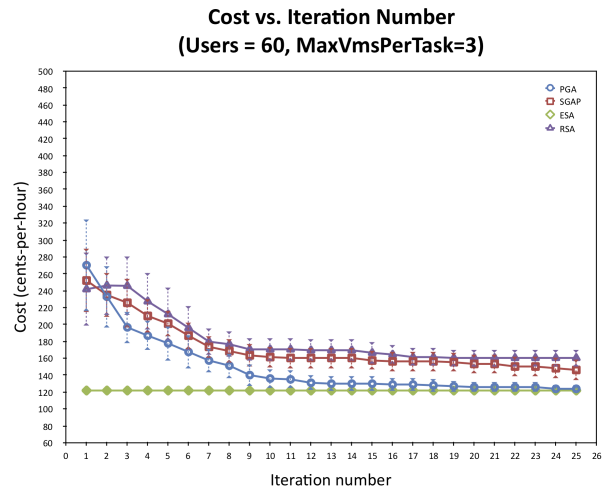
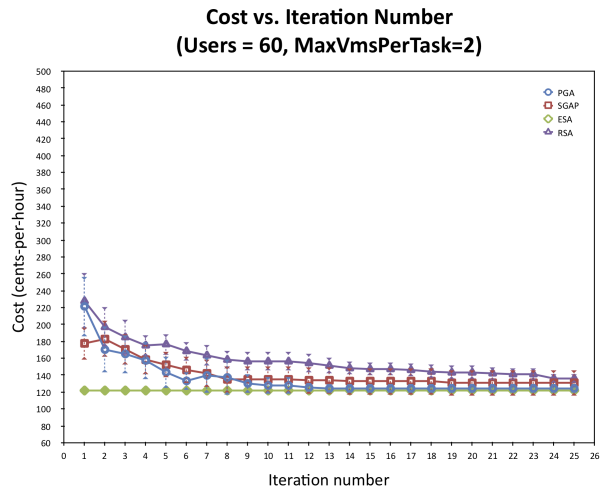


Figure C.2: Cost vs. Iterations (Users=60)

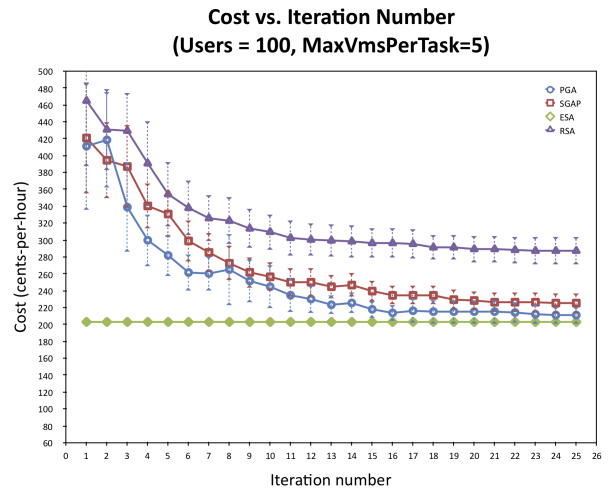
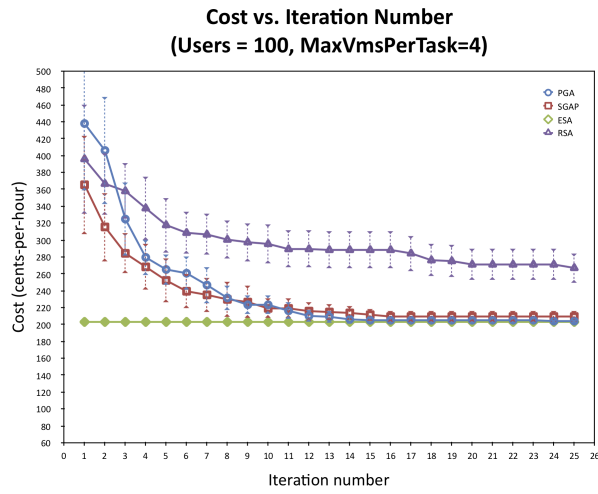
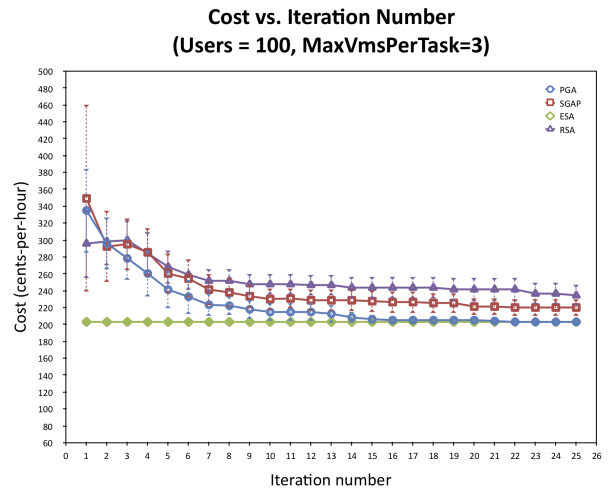
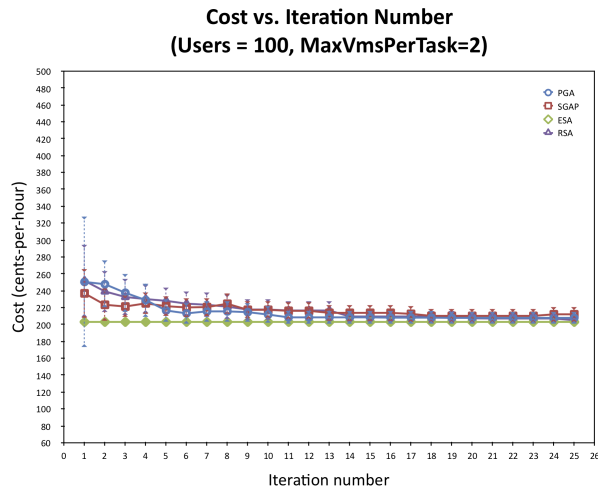


Figure C.3: Cost vs. Iterations (Users=100)

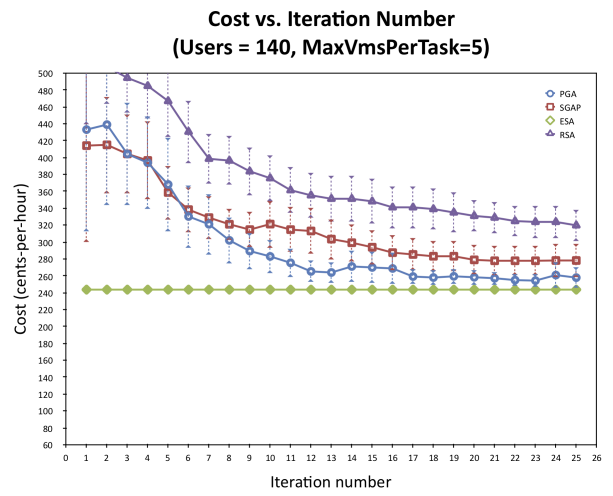
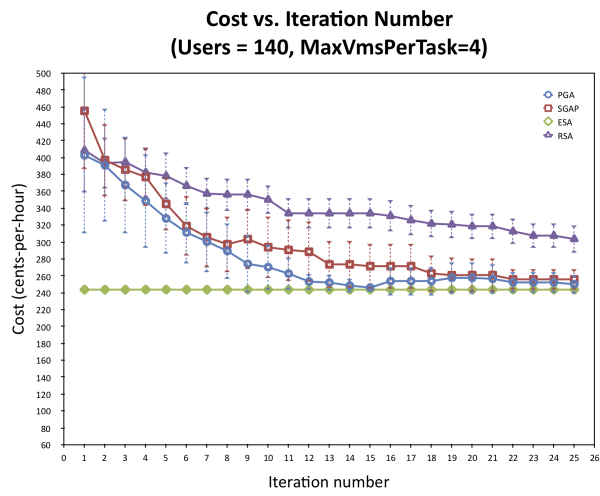
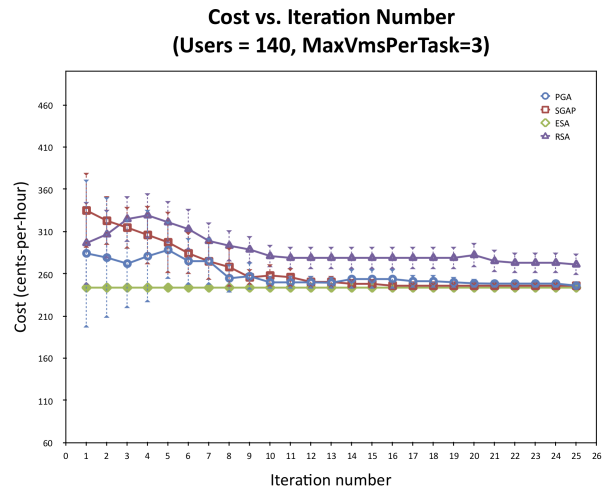
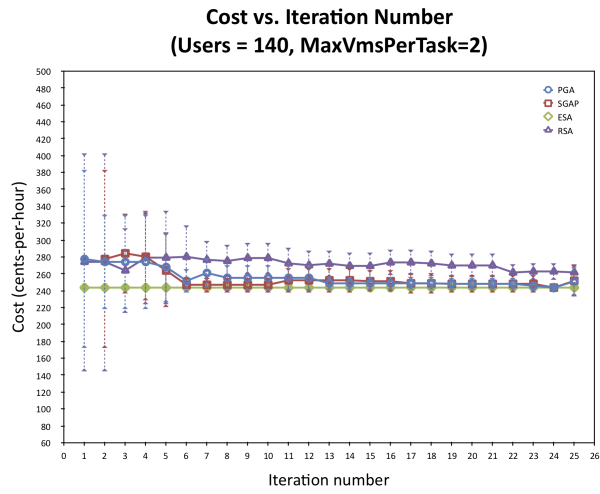


Figure C.4: Cost vs. Iterations (Users=140)

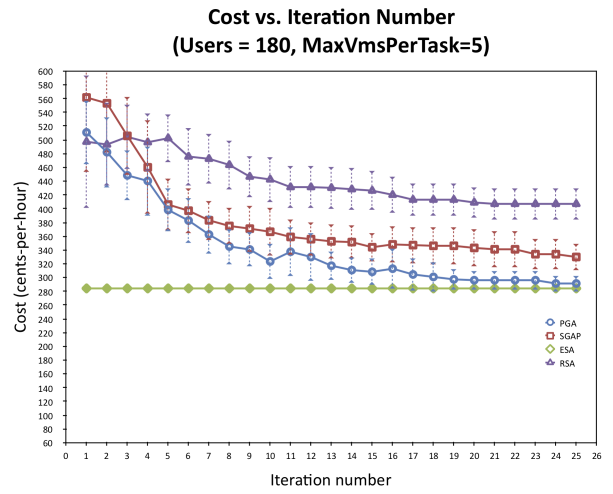
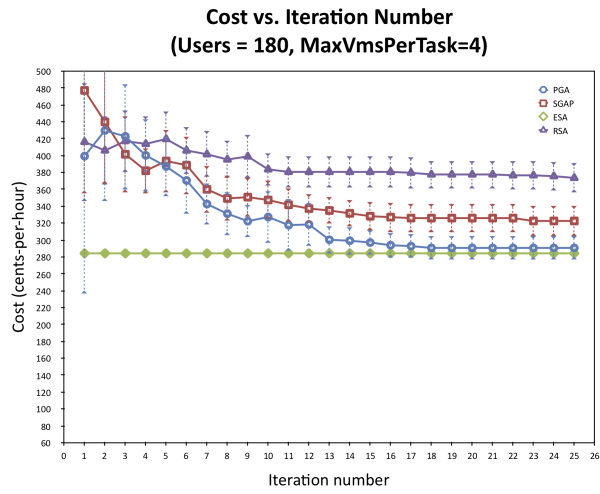
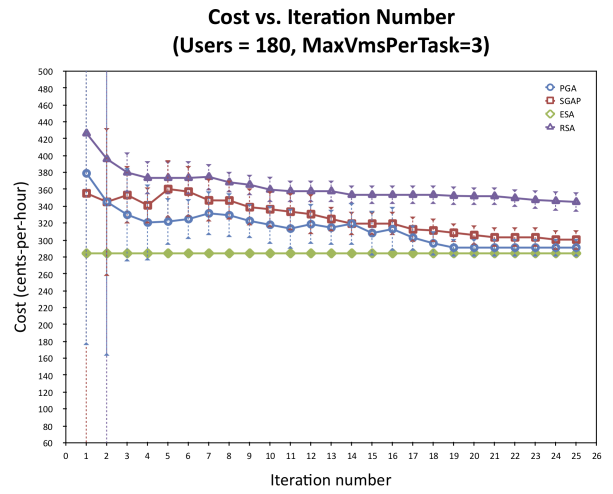
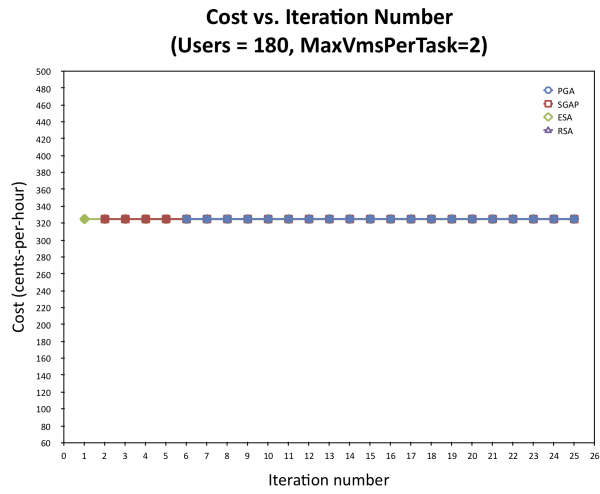


Figure C.5: Cost vs. Iterations (Users=180)

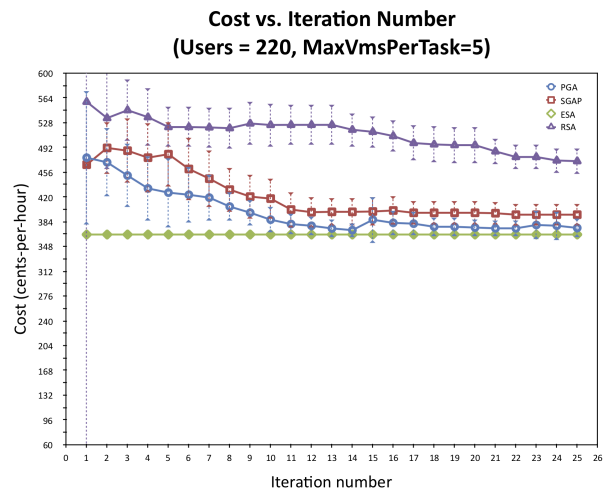
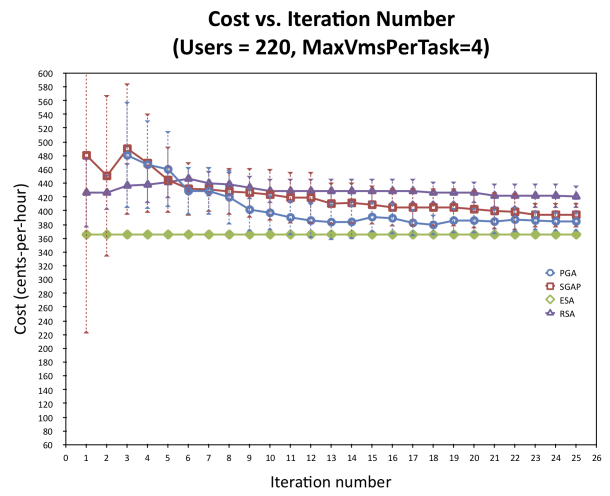
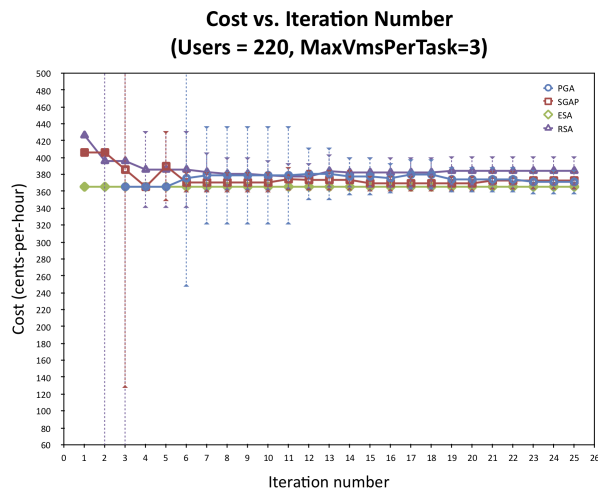


Figure C.6: Cost vs. Iterations (Users=220)



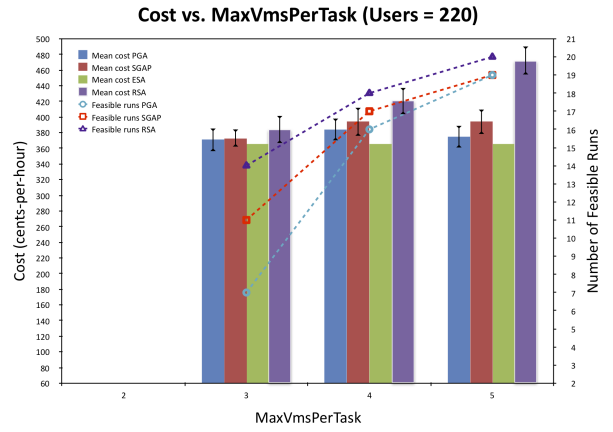
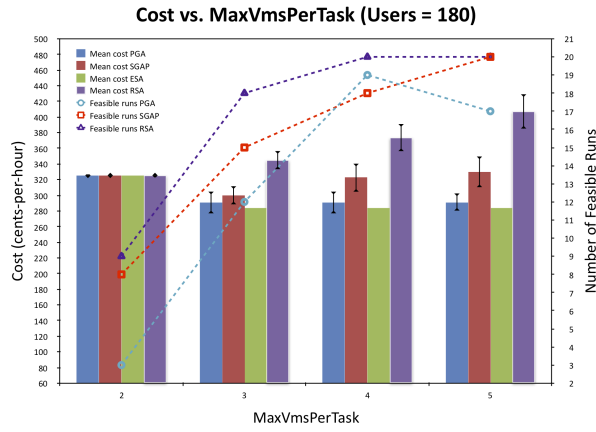
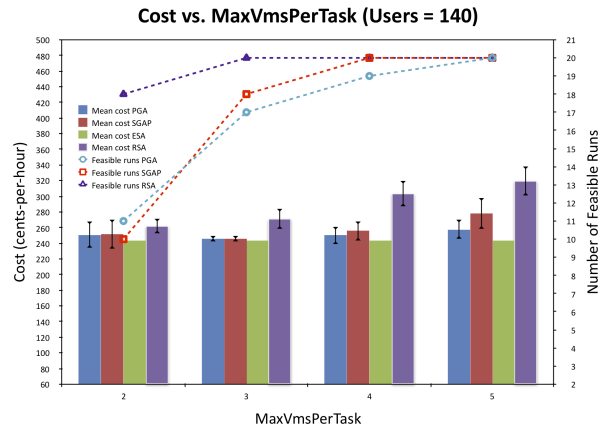
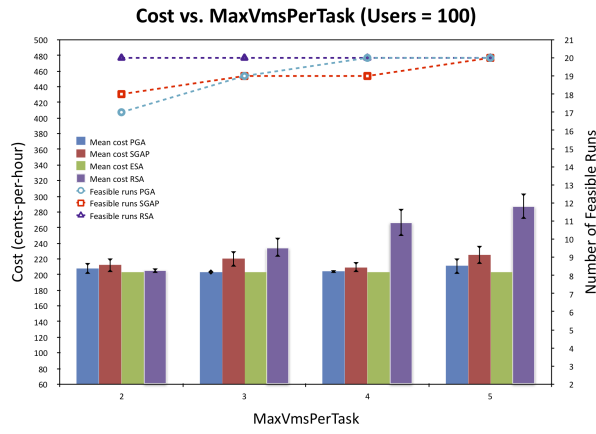
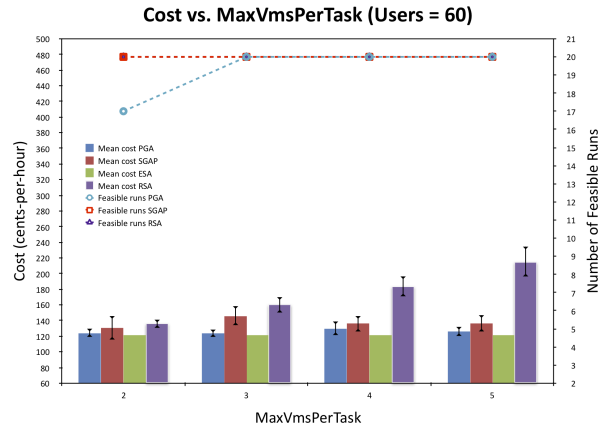
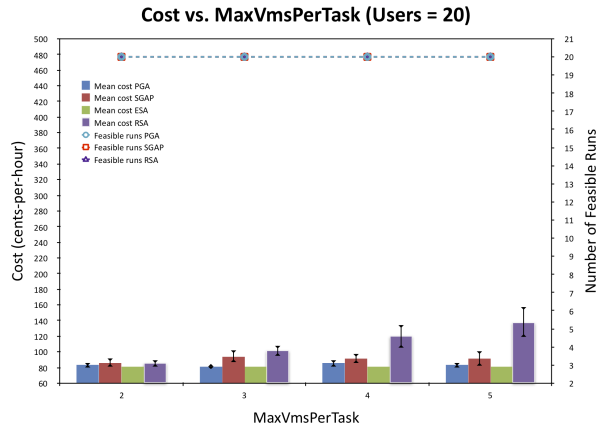


Figure C.7: Cost vs. MaxVmsPerTask

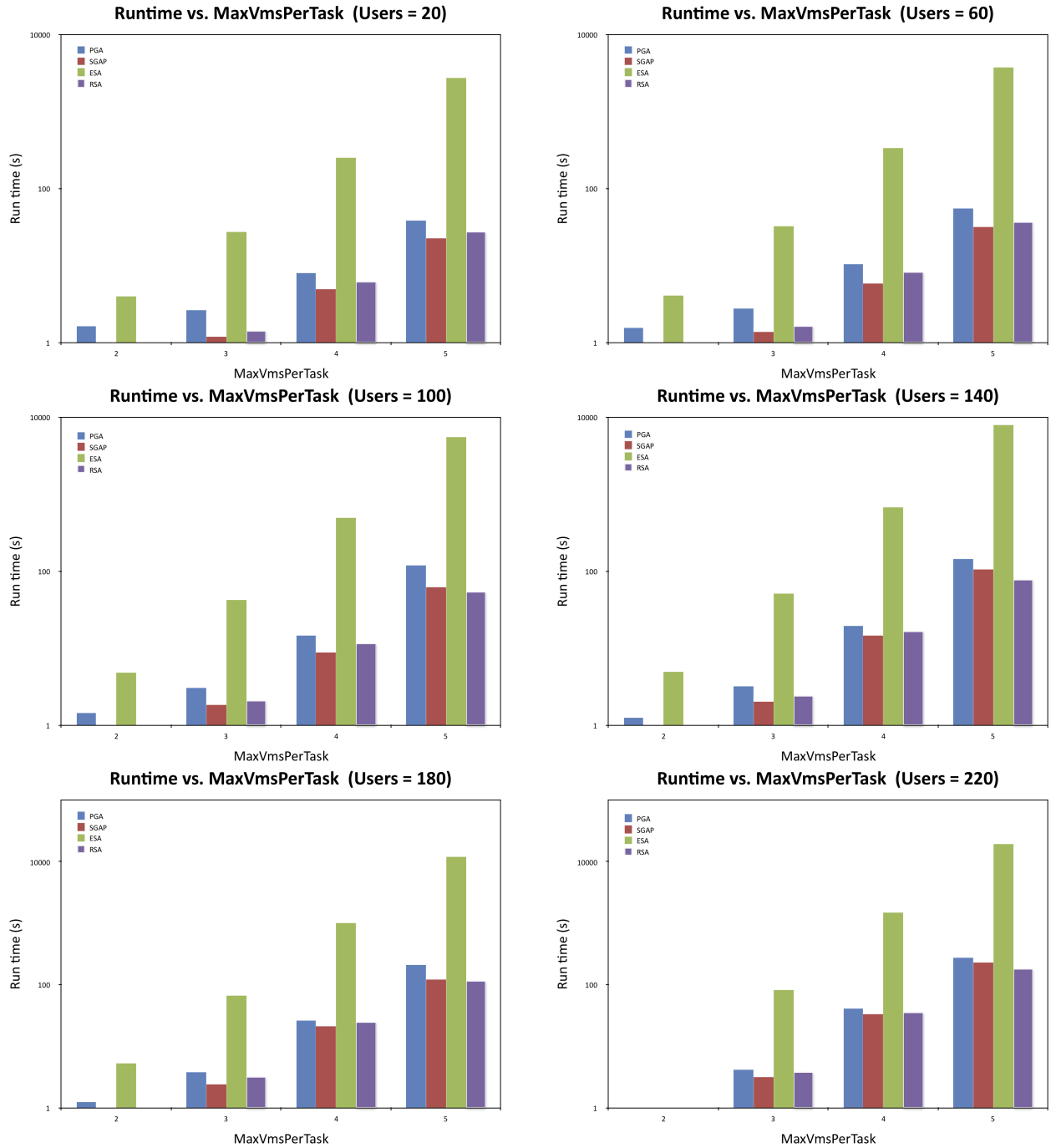


Figure C.8: Runtime vs. MaxVmsPerTask. The runtime is of one complete run and not of 20 runs. Runtime is in logarithmic scale.

# Bibliography

- [1] We Are Social and Hootsuite. (2018, Jan) Digital in 2018. [Online]. Available: <https://digitalreport.wearesocial.com/>
- [2] T. Velte, A. Velte, and R. Elsenpeter, *Cloud computing: a practical approach*. McGraw-Hill Osborne Media, 2009.
- [3] R. Ranjan, L. Zhao, X. Wu, A. Liu, A. Quiroz, and M. Parashar, “Peer-to-peer cloud provisioning: Service discovery and load-balancing,” in *Cloud Computing*, ser. Computer Communications and Networks, N. Antonopoulos and L. Gillam, Eds. Springer London, 2010, pp. 195–217.
- [4] Q. Zhang, L. Cheng, and R. Boutaba, “Cloud computing: state-of-the-art and research challenges,” *Journal of Internet Services and Applications*, vol. 1, no. 1, pp. 7–18, 2010. [Online]. Available: <http://dx.doi.org/10.1007/s13174-010-0007-6>
- [5] “eCommerce web site performance today: An updated look at consumer reaction to a poor online shopping experience,” White Paper, Forrester Consulting, pp. 1–21, Aug. 2009.
- [6] Amazon Web Services. (2019) Amazon EC2 Auto Scaling. [Online]. Available: <https://aws.amazon.com/ec2/autoscaling/>
- [7] Microsoft Azure. (2018) Overview of autoscale in Microsoft Azure Virtual Machines, Cloud Services, and Web Apps. [Online]. Available: <https://docs.microsoft.com/en-us/azure/azure-monitor/platform/autoscale-overview>
- [8] 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.
- [9] O. Ibidunmoye, M. H. Moghadam, E. B. Lakew, and E. Elmroth, “Adaptive service performance control using cooperative fuzzy reinforcement learning in virtualized

- environments,” in *Proceedings of the 10th International Conference on Utility and Cloud Computing*, ser. UCC '17. New York, NY, USA: ACM, 2017, pp. 19–28.
- [10] P. Jamshidi, A. Sharifloo, C. Pahl, H. Arabnejad, A. Metzger, and G. Estrada, “Fuzzy self-learning controllers for elasticity management in dynamic cloud architectures,” in *2016 12th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA)*, April 2016, pp. 70–79.
- [11] Jeff Barr. (2018) New - Predictive Scaling for EC2, Powered by Machine Learning. [Online]. Available: <https://aws.amazon.com/blogs/aws/new-predictive-scaling-for-ec2-powered-by-machine-learning/>
- [12] B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal, and T. Wood, “Agile dynamic provisioning of multi-tier internet applications,” *ACM Trans. on Autonomous and Adaptive Systems*, vol. 3, no. 1, 2008, 39 pages.
- [13] W. Iqbal, M. N. Dailey, D. Carrera, and P. Janecek, “Adaptive resource provisioning for read intensive multi-tier applications in the cloud,” *Future Generation Computer Systems*, vol. 27, no. 6, pp. 871 – 879, 2011. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S0167739X10002098>
- [14] I. Gergin, B. Simmons, and M. Litoiu, “A decentralized autonomic architecture for performance control in the cloud,” in *2014 IEEE International Conference on Cloud Engineering*. IEEE, 2014, pp. 574–579.
- [15] Y. Shoaib and O. Das, “Using layered bottlenecks for virtual machine provisioning in the clouds,” in *Proceedings of the 2012 IEEE/ACM Fifth International Conference on Utility and Cloud Computing (UCC 2012)*, 5–8 Nov. 2012, pp. 109–116,  
© 2012 IEEE. Reprinted, with permission from Y. Shoaib and O. Das, “Using Layered Bottlenecks for Virtual Machine Provisioning in the Clouds”, UCC 2012, Nov 2012.
- [16] G. Franks, D. Petriu, M. Woodside, J. Xu, and P. Tregunno, “Layered bottlenecks and their mitigation,” in *3rd Int. Conf. on Quantitative Evaluation of Systems*, 2006, pp. 103–114.

- [17] R. Han, M. M. Ghanem, L. Guo, Y. Guo, and M. Osmond, “Enabling cost-aware and adaptive elasticity of multi-tier cloud applications,” *Future Generation Computer Systems*, vol. 32, pp. 82–98, 2014.
- [18] G. Franks, P. Maly, M. Woodside, D. Petriu, and A. Hubbard, *Layered Queueing Network Solver and Simulator User Manual*, Real-time and Distributed Systems Lab, Carleton University, Ottawa, Jan. 2013. [Online]. Available: <http://www.sce.carleton.ca/rads/lqns/LQNSUserMan-jan13.pdf>
- [19] R. G. Franks, “Performance analysis of distributed server systems,” Ph.D. dissertation, Ottawa, Ont., Canada, Canada, 1999.
- [20] T.-K. Liu, S. Kumaran, and J.-Y. Chung, “Performance engineering of a java-based ecommerce system,” in *Proceedings of the 2004 IEEE International Conference on e-Technology, e-Commerce and e-Service (EEE’04)*, ser. EEE ’04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 33–37. [Online]. Available: <http://dl.acm.org/citation.cfm?id=987681.987818>
- [21] A. Ufimtsev and L. Murphy, “Performance modeling of a javaee component application using layered queuing networks: revised approach and a case study,” in *Proceedings of the 2006 conference on Specification and verification of component-based systems*, ser. SAVCBS ’06. New York, NY, USA: ACM, 2006, pp. 11–18. [Online]. Available: <http://doi.acm.org/10.1145/1181195.1181198>
- [22] Y. Shoaib and O. Das, “Web application performance modeling using layered queueing networks,” *Electronic Notes in Theoretical Computer Science*, vol. 275, pp. 123–142, Sep. 2011, (Fifth International Workshop on the Practical Application of Stochastic Modelling (PASM)). [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S1571066111001009>
- [23] T. Omari, G. Franks, M. Woodside, and A. Pan, “Solving layered queueing networks of large client-server systems with symmetric replication,” in *5th Int. workshop on Software and performance*, 2005, pp. 159–166.

- [24] P. Mell and T. Grance, "The NIST definition of cloud computing," NIST (National Institute of Standards and Technology), Gaithersburg, MD 20899-8930, Special Publication 800-145, Sep 2011. [Online]. Available: <http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf>
- [25] S. Islam, K. Lee, A. Fekete, and A. Liu, "How a consumer can measure elasticity for cloud platforms," in *Proceedings of the third joint WOSP/SIPEW international conference on Performance Engineering*, ser. ICPE '12. New York, NY, USA: ACM, 2012, pp. 85–96. [Online]. Available: <http://doi.acm.org/10.1145/2188286.2188301>
- [26] D. Owens, "Securing elasticity in the cloud," *Commun. ACM*, vol. 53, no. 6, pp. 46–51, Jun. 2010. [Online]. Available: <http://doi.acm.org/10.1145/1743546.1743565>
- [27] R. Figueiredo, P. Dinda, and J. Fortes, "Guest editors' introduction: Resource virtualization renaissance," *Computer*, vol. 38, no. 5, pp. 28–31, may 2005.
- [28] R. Uhlig, G. Neiger, D. Rodgers, A. Santoni, F. Martins, A. Anderson, S. Bennett, A. Kagi, F. Leung, and L. Smith, "Intel virtualization technology," *Computer*, vol. 38, no. 5, pp. 48–56, may 2005.
- [29] M. Rosenblum and T. Garfinkel, "Virtual machine monitors: current technology and future trends," *Computer*, vol. 38, no. 5, pp. 39–47, may 2005.
- [30] Q. Liu, C. Weng, M. Li, and Y. Luo, "An in-vm measuring framework for increasing virtual machine security in clouds," *Security Privacy, IEEE*, vol. 8, no. 6, pp. 56–62, nov.-dec. 2010.
- [31] T. Killalea, "Meet the virts," *Queue*, vol. 6, no. 1, pp. 14–18, Jan. 2008. [Online]. Available: <http://doi.acm.org/10.1145/1348583.1348589>
- [32] M. Price, "The paradox of security in virtual environments," *Computer*, vol. 41, no. 11, pp. 22–28, nov. 2008.
- [33] H.-Y. Tsai, M. Siebenhaar, A. Miede, Y. Huang, and R. Steinmetz, "Threat as a service?: Virtualization's impact on cloud security," *IT Professional*, vol. 14, no. 1, pp. 32–37, jan.-feb. 2012.

- [34] C. U. Smith and L. G. Williams, *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 2002.
- [35] D. Peng, Y. Yuan, K. Yue, X. Wang, and A. Zhou, "Capacity planning for composite web services using queueing network-based models," in *Advances in Web-Age Information Management*, ser. Lecture Notes in Computer Science, Q. Li, G. Wang, and L. Feng, Eds. Springer Berlin Heidelberg, 2004, vol. 3129, pp. 439–448. [Online]. Available: [http://dx.doi.org/10.1007/978-3-540-27772-9\\_44](http://dx.doi.org/10.1007/978-3-540-27772-9_44)
- [36] D. Menasce, "Load testing of web sites," *Internet Computing, IEEE*, vol. 6, no. 4, pp. 70–74, 2002.
- [37] Y. Shoaib, "Performance measurement and analytic modeling of a web application," Master's thesis, Ryerson University, Toronto, Ontario, Canada, May 2011.
- [38] A. Gambi, G. Toffetti, and S. Comai, "Model-driven web engineering performance prediction with layered queue networks," in *International Conference on Web Engineering*. Springer, 2010, pp. 25–36.
- [39] S. Kounev and A. P. Buchmann, "Performance modeling and evaluation of large-scale j2ee applications," in *Int. CMG Conference*, vol. 11, 2003, pp. 273–283.
- [40] B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer, and A. Tantawi, "Analytic modeling of multitier internet applications," *ACM Trans. Web*, vol. 1, no. 1, May 2007. [Online]. Available: <http://doi.acm.org/10.1145/1232722.1232724>
- [41] G. Bolch, S. Greiner, H. de Meer, and K. S. Trivedi, *Queueing Networks and Markov Chains: Modeling and Performance Evaluation with Computer Science Applications*. New York, NY, USA: Wiley-Interscience, 2005.
- [42] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik, *Quantitative system performance: computer system analysis using queueing network models*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1984.

- [43] X. P. Wu, "An approach to predicting performance for component based systems," Master's thesis, Carleton University, Ottawa, Ontario, Canada, Jul. 2003.
- [44] T. A. Israr, D. H. Lau, G. Franks, and M. Woodside, "Automatic generation of layered queuing software performance models from commonly available traces," in *Proceedings of the 5th international workshop on Software and performance*, ser. WOSP '05. New York, NY, USA: ACM, 2005, pp. 147–158. [Online]. Available: <http://doi.acm.org/10.1145/1071021.1071037>
- [45] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2009.
- [46] C. R. Reeves and J. E. Rowe, *Genetic Algorithms: Principles and Perspectives: A Guide to GA Theory*. Norwell, MA, USA: Kluwer Academic Publishers, 2002.
- [47] M. Srinivas and L. M. Patnaik, "Genetic algorithms: a survey," *Computer*, vol. 27, no. 6, pp. 17–26, June 1994.
- [48] J. Holland, "Adaptation in artificial and natural systems," *Ann Arbor: The University of Michigan Press*, 1975.
- [49] U. Mehboob, J. Qadir, S. Ali, and A. Vasilakos, "Genetic algorithms in wireless networking: Techniques, applications, and issues," *Soft Comput.*, vol. 20, no. 6, pp. 2467–2501, Jun. 2016. [Online]. Available: <http://dx.doi.org/10.1007/s00500-016-2070-9>
- [50] R. N. Calheiros, R. Ranjan, and R. Buyya, "Virtual machine provisioning based on analytical performance and QoS in cloud computing environments," in *Proc. of the 2011 Int. Conf. on Parallel Processing*, 2011, pp. 295–304.
- [51] Y. Shoaib and O. Das, "Performance-oriented cloud provisioning: Taxonomy and survey," *arXiv preprint arXiv:1411.5077*, 2014.
- [52] Amazon Web Services. (2019) Amazon EC2. [Online]. Available: <https://aws.amazon.com/ec2/>



- [53] Google Cloud. (2019) Compute Engine. [Online]. Available: <https://cloud.google.com/compute/>
- [54] D. Armour. (2019) Azure Stack IaaS - part one. [Online]. Available: <https://azure.microsoft.com/en-us/blog/azure-stack-iaas-part-one/>
- [55] Google Cloud. (2019) Kubernetes Engine. [Online]. Available: <https://cloud.google.com/kubernetes-engine/>
- [56] Microsoft. (2019) Azure Kubernetes Service (AKS). [Online]. Available: <https://azure.microsoft.com/en-ca/services/kubernetes-service/>
- [57] Amazon Web Services. (2019) Amazon Elastic Container Service. [Online]. Available: <https://aws.amazon.com/ecs/>
- [58] M. Lukša, *Kubernetes in Action*. Manning Publications Company, 2018.
- [59] H. Nguyen Van, F. Dang Tran, and J.-M. Menaud, “Autonomic virtual resource management for service hosting platforms,” in *Proceedings of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing*, ser. CLOUD '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–8. [Online]. Available: <http://dx.doi.org/10.1109/CLOUD.2009.5071526>
- [60] W. Iqbal, M. Dailey, and D. Carrera, “SLA-driven dynamic resource management for multi-tier web applications in a cloud,” in *2010 10th IEEE/ACM Int. Conf. on Cluster, Cloud and Grid Computing*. IEEE, 2010, pp. 832–837.
- [61] A. Quiroz, H. Kim, M. Parashar, N. Gnanasambandam, and N. Sharma, “Towards autonomic workload provisioning for enterprise grids and clouds,” in *Grid Computing, 2009 10th IEEE/ACM International Conference on*, 2009, pp. 50–57.
- [62] L. M. Vaquero, L. Roderio-Merino, and R. Buyya, “Dynamically scaling applications in the cloud,” *SIGCOMM Comput. Commun. Rev.*, vol. 41, no. 1, pp. 45–52, Jan. 2011. [Online]. Available: <http://doi.acm.org/10.1145/1925861.1925869>

- [63] G. Kroah-Hartman, "Kernel korner: Hot plug," *Linux J.*, vol. 2002, no. 96, p. 7, Apr. 2002. [Online]. Available: <http://dl.acm.org/citation.cfm?id=513180.513187>
- [64] S. Panneerselvam and M. M. Swift, "Dynamic processors demand dynamic operating systems," in *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*, ser. HotPar'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–7. [Online]. Available: [http://static.usenix.org/event/hotpar10/tech/full\\_papers/Panneerselvam.pdf](http://static.usenix.org/event/hotpar10/tech/full_papers/Panneerselvam.pdf)
- [65] Z. Mwaikambo, A. Raj, R. Russell, J. Schopp, and S. Vaddagiri, "Linux kernel hotplug cpu support," in *Proceedings of the Linux Symposium*, 2004, pp. 469–481. [Online]. Available: [http://www.linuxsymposium.org/archives/OLS/Reprints-2004/LinuxSymposium2004\\_All.pdf](http://www.linuxsymposium.org/archives/OLS/Reprints-2004/LinuxSymposium2004_All.pdf)
- [66] S. Panneerselvam and M. M. Swift, "Chameleon: operating system support for dynamic processors," in *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVII. New York, NY, USA: ACM, 2012, pp. 99–110. [Online]. Available: <http://doi.acm.org/10.1145/2150976.2150988>
- [67] L. Yazdanov and C. Fetzer, "Vertical scaling for prioritized vms provisioning," in *Proceedings of the 2012 Second International Conference on Cloud and Green Computing*, ser. CGC '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 118–125. [Online]. Available: <http://dx.doi.org/10.1109/CGC.2012.108>
- [68] VMware. (2014) VMware vSphere: Server Virtualization, Cloud Infrastructure. [Online]. Available: <http://www.vmware.com/products/vsphere>
- [69] M. OTEY, "New features in vsphere 4.0." *Windows IT Pro*, vol. 15, no. 12, p. 19, 2009.
- [70] "VMware vSphere 5 Competitive Reviewers Guide," White Paper, VMware, Inc., pp. 1–34, 2011. [Online]. Available: <https://www.vmware.com/files/pdf/VMware-vSphere-Competitive-Reviewers-guide-WP-EN.pdf>
- [71] Y. Shoaib and O. Das, "Modeling website workload using neural networks," *arXiv preprint arXiv:1507.07204*, 2015.

- [72] Amazon Web Services, Inc. or its affiliates. (2014) Auto scaling. [Online]. Available: <http://aws.amazon.com/autoscaling/>
- [73] Google Cloud. (2019) Autoscaling Groups of Instances. [Online]. Available: <https://cloud.google.com/compute/docs/autoscaler/>
- [74] C. Barclay. (2019) Automatic Scaling with Amazon ECS. [Online]. Available: <https://aws.amazon.com/blogs/compute/automatic-scaling-with-amazon-ecs/>
- [75] Google Cloud. (2019) Scaling an application. [Online]. Available: <https://cloud.google.com/kubernetes-engine/docs/how-to/scaling-apps>
- [76] Google Cloud. (2019) Creating Groups of Managed Instances. [Online]. Available: <https://cloud.google.com/compute/docs/instance-groups/creating-groups-of-managed-instances>
- [77] Google Cloud. (2019) Deploying a stateful application. [Online]. Available: <https://cloud.google.com/kubernetes-engine/docs/how-to/stateful-apps>
- [78] Google Cloud. (2019) Deploying a stateless application. [Online]. Available: <https://cloud.google.com/kubernetes-engine/docs/how-to/stateless-apps>
- [79] The Kubernetes Authors. (2019) Pod Overview - Kubernetes. [Online]. Available: <https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/>
- [80] R. Ranganathan, “A highly-available and scalable microservice architecture for access management,” Master’s thesis, Aalto University, School of Science, Espoo, Finland, Sep. 2018.
- [81] Canonical Ltd. (2014) MAAS. [Online]. Available: <https://maas.ubuntu.com/>
- [82] R. Walters, “Maas effect: Canonical dives into bare metal cluster provisioning,” *ExtremeTech.com*, Apr 04 2012, copyright (c) 2012 Ziff Davis Media Inc. All rights reserved. Last updated - 2012-04-05.
- [83] P. Costa, M. Migliavacca, P. Pietzuch, and A. L. Wolf, “NaaS: network-as-a-service in the cloud,” in *Proceedings of the 2nd USENIX conference on Hot Topics in Management of*

- Internet, Cloud, and Enterprise Networks and Services*, ser. Hot-ICE'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 1–1. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2228283.2228285>
- [84] A. W. Services, *Amazon Elastic Compute Cloud: User Guide for Linux (API Version 2014-10-01)*, Amazon AWS. [Online]. Available: <http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-ug.pdf>
- [85] Microsoft. (2014) Azure Regions. [Online]. Available: <http://azure.microsoft.com/en-us/regions/>
- [86] J. Chinneck, M. Litoiu, and M. Woodside, “Real-time multi-cloud management needs application awareness,” in *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '14. New York, NY, USA: ACM, 2014, pp. 293–296. [Online]. Available: <http://doi.acm.org/10.1145/2568088.2576763>
- [87] A. Faisal, D. Petriu, and M. Woodside, “Network latency impact on performance of software deployed across multiple clouds,” in *Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research*, ser. CASCON '13. Riverton, NJ, USA: IBM Corp., 2013, pp. 216–229. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2555523.2555547>
- [88] H. Benfenatki, C. F. D. Silva, A. N. Benharkat, and P. Ghodous, “Cloud application development methodology,” in *Proceedings of the 2014 IEEE/WIC/ACM International Joint Conferences on Web Intelligence (WI) and Intelligent Agent Technologies (IAT) - Volume 01*, ser. WI-IAT '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 13–20. [Online]. Available: <http://dx.doi.org/10.1109/WI-IAT.2014.11>
- [89] N. Ferry, A. Rossini, F. Chauvel, B. Morin, and A. Solberg, “Towards model-driven provisioning, deployment, monitoring, and adaptation of multi-cloud systems,” in *Proceedings of the 2013 IEEE Sixth International Conference on Cloud Computing*, ser. CLOUD '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 887–894. [Online]. Available: <http://dx.doi.org/10.1109/CLOUD.2013.133>

- [90] J. Chinneck, M. Litoiu, and M. Woodside, "Real-time multi-cloud management needs application awareness," in *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '14. New York, NY, USA: ACM, 2014, pp. 293–296. [Online]. Available: <http://doi.acm.org/10.1145/2568088.2576763>
- [91] T. Zheng, "Model-based dynamic resource management for multi tier information systems," Ph.D. dissertation, Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Canada, Aug 2007.
- [92] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, and R. Buyya, "Cloudsim: A toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Softw. Pract. Exper.*, vol. 41, no. 1, pp. 23–50, Jan. 2011. [Online]. Available: <http://dx.doi.org/10.1002/spe.995>
- [93] J. Z. Li, M. Woodside, J. Chinneck, and M. Litoiu, "Cloudopt: Multi-goal optimization of application deployments across a cloud," in *Proceedings of the 7th International Conference on Network and Services Management*, ser. CNSM '11. Laxenburg, Austria, Austria: International Federation for Information Processing, 2011, pp. 162–170. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2147671.2147697>
- [94] J. Li, M. Woodside, J. Chinneck, and M. Litiou, "Adaptive cloud deployment using persistence strategies and application awareness," *IEEE Transactions on Cloud Computing*, vol. 5, no. 2, pp. 277–290, April 2017.
- [95] N. Huber, F. Brosig, S. Spinner, S. Kounev, and M. Bähr, "Model-based self-aware performance and resource management using the descartes modeling language," *IEEE Transactions on Software Engineering*, vol. 43, no. 5, pp. 432–452, May 2017.
- [96] D. Menasce, "Automatic QoS Control," *IEEE Internet Computing*, vol. 7, no. 1, pp. 92–95, 2003.
- [97] R. Buyya, R. Ranjan, and R. Calheiros, "Intercloud: Utility-oriented federation of cloud computing environments for scaling of application services," in *Algorithms and Architectures for Parallel Processing*, ser. Lecture Notes in Computer Science, C.-H. Hsu, L. Yang, J. Park,

- and S.-S. Yeo, Eds. Springer Berlin Heidelberg, 2010, vol. 6081, pp. 13–31. [Online]. Available: [http://dx.doi.org/10.1007/978-3-642-13119-6\\_2](http://dx.doi.org/10.1007/978-3-642-13119-6_2)
- [98] M. Stillwell, D. Schanzenbach, F. Vivien, and H. Casanova, “Resource allocation algorithms for virtualized service hosting platforms,” *J. Parallel Distrib. Comput.*, vol. 70, no. 9, pp. 962–974, Sep. 2010. [Online]. Available: <http://dx.doi.org/10.1016/j.jpdc.2010.05.006>
- [99] D. Pandit, S. Chattopadhyay, M. Chattopadhyay, and N. Chaki, “Resource allocation in cloud using simulated annealing,” in *Applications and Innovations in Mobile Computing (AIMoC)*, 2014, Feb 2014, pp. 21–27.
- [100] B. URGANONKAR, A. L. ROSENBERG, and P. SHENOY, “Application placement on a cluster of servers,” *International Journal of Foundations of Computer Science*, vol. 18, no. 05, pp. 1023–1041, 2007. [Online]. Available: <http://www.worldscientific.com/doi/abs/10.1142/S012905410700511X>
- [101] Z. Xiao, W. Song, and Q. Chen, “Dynamic resource allocation using virtual machines for cloud computing environment,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 24, no. 6, pp. 1107–1117, Jun. 2013. [Online]. Available: <http://dx.doi.org/10.1109/TPDS.2012.283>
- [102] S. Chabaa, A. Zeroual, and J. Antari, “Identification and prediction of internet traffic using artificial neural networks,” *Journal of Intelligent Learning Systems and Applications*, vol. 2, no. 3, pp. 147–155, August 2010.
- [103] J. Prevost, K. Nagothu, B. Kelley, and M. Jamshidi, “Prediction of cloud data center networks loads using stochastic and neural models,” in *System of Systems Engineering (SoSE), 2011 6th International Conference on*, June 2011, pp. 276–281.
- [104] M. Mishra, A. Das, P. Kulkarni, and A. Sahoo, “Dynamic resource management using virtual machine migrations,” *Communications Magazine, IEEE*, vol. 50, no. 9, pp. 34–40, 2012.
- [105] N. Calcavecchia, O. Biran, E. Hadad, and Y. Moatti, “Vm placement strategies for cloud scenarios,” in *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, June 2012, pp. 852–859.

- [106] J. Li, J. Chinneck, M. Woodside, M. Litoiu, and G. Iszlai, "Performance model driven QoS guarantees and optimization in clouds," in *2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing*, 2009, pp. 15–22.
- [107] N. Huber, F. Brosig, and S. Kounev, "Model-based self-adaptive resource allocation in virtualized environments," in *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS '11)*. ACM, 2011, pp. 90–99.
- [108] A. Das, T. Adhikary, M. Razzaque, and C. S. Hong, "An intelligent approach for virtual machine and qos provisioning in cloud computing," in *Information Networking (ICOIN), 2013 International Conference on*, Jan 2013, pp. 462–467.
- [109] D. A. Menasce and V. Almeida, *Capacity Planning for Web Services: Metrics, Models, and Methods*, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2001.
- [110] D. A. Menasce, L. W. Dowdy, and V. A. F. Almeida, *Performance by Design: Computer Capacity Planning By Example*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2004.
- [111] J. Dille, R. Friedrich, T. Jin, and J. Rolia, "Web server performance measurement and modeling techniques," *Perform. Eval.*, vol. 33, no. 1, pp. 5–26, Jun. 1998. [Online]. Available: [http://dx.doi.org/10.1016/S0166-5316\(98\)00008-X](http://dx.doi.org/10.1016/S0166-5316(98)00008-X)
- [112] A. Karve, T. Kimbrel, G. Pacifici, M. Spreitzer, M. Steinder, M. Sviridenko, and A. Tantawi, "Dynamic placement for clustered web applications," in *Proceedings of the 15th International Conference on World Wide Web*, ser. WWW '06. New York, NY, USA: ACM, 2006, pp. 595–604. [Online]. Available: <http://doi.acm.org/10.1145/1135777.1135865>
- [113] T. Kimbrel, M. Steinder, M. Sviridenko, and A. Tantawi, "Dynamic application placement under service and memory constraints," in *Proceedings of the 4th International Conference on Experimental and Efficient Algorithms*, ser. WEA'05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 391–402. [Online]. Available: [http://dx.doi.org/10.1007/11427186\\_34](http://dx.doi.org/10.1007/11427186_34)
- [114] W. Iqbal, M. Dailey, and D. Carrera, "Sla-driven adaptive resource management for web applications on a heterogeneous compute cloud," in *Proceedings of the 1st International*

- Conference on Cloud Computing*, ser. CloudCom '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 243–253. [Online]. Available: [http://dx.doi.org/10.1007/978-3-642-10665-1\\_22](http://dx.doi.org/10.1007/978-3-642-10665-1_22)
- [115] E. Kijssipongse and S. Vannarat, “Autonomic resource provisioning in rocks clusters using eucalyptus cloud computing,” in *Int. Conf. on Management of Emergent Digital EcoSystems*. ACM, 2010, pp. 61–66.
- [116] G. Franks, T. Al-Omari, M. Woodside, O. Das, and S. Derisavi, “Enhanced modeling and solution of layered queueing networks,” *IEEE Trans. Softw. Eng.*, vol. 35, no. 2, pp. 148–161, Mar. 2009. [Online]. Available: <http://dx.doi.org/10.1109/TSE.2008.74>
- [117] H. C. Lim, S. Babu, J. S. Chase, and S. S. Parekh, “Automated control in cloud computing: Challenges and opportunities,” in *Proceedings of the 1st Workshop on Automated Control for Datacenters and Clouds*, ser. ACDC '09. New York, NY, USA: ACM, 2009, pp. 13–18. [Online]. Available: <http://doi.acm.org/10.1145/1555271.1555275>
- [118] R. Chi, Z. Qian, and S. Lu, “A heuristic approach for scalability of multi-tiers web application in clouds,” in *Proceedings of the 2011 Fifth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, ser. IMIS '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 28–35. [Online]. Available: <http://dx.doi.org/10.1109/IMIS.2011.80>
- [119] E. Casalicchio, D. A. Menascé, and A. Aldhalaan, “Autonomic resource provisioning in cloud systems with availability goals,” in *Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference*, ser. CAC '13. New York, NY, USA: ACM, 2013, pp. 1:1–1:10. [Online]. Available: <http://doi.acm.org/10.1145/2494621.2494623>
- [120] G. Jung, K. Joshi, M. Hiltunen, R. Schlichting, and C. Pu, “Performance and availability aware regeneration for cloud based multitier applications,” in *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*, 2010, pp. 497–506.
- [121] A. Beloglazov, J. Abawajy, and R. Buyya, “Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing,” *Future Gener. Comput. Syst.*,



- vol. 28, no. 5, pp. 755–768, May 2012. [Online]. Available: <http://dx.doi.org/10.1016/j.future.2011.04.017>
- [122] M. Mroz and G. Franks, “A performance experiment system supporting fast mapping of system issues,” in *Proceedings of the Fourth International ICST Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS '09)*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2009, 10 pages.
- [123] M. Mroz and G. Franks, “A performance experiment system supporting fast mapping of system issues,” in *4th Int. ICST Conf. on Performance Evaluation Methodologies and Tools*, 2009, 10 pages.
- [124] Y. Shoaib and O. Das, “Cloud VM provisioning using analytical performance models,” in *Proceedings of the 2019 IEEE Cloud*, Accepted as a short paper.
- [125] K. Brown and B. Woolf, “Implementation patterns for microservices architectures,” in *Proceedings of the 23rd Conference on Pattern Languages of Programs*, ser. PLoP '16. USA: The Hillside Group, 2016, pp. 7:1–7:35. [Online]. Available: <http://dl.acm.org/citation.cfm?id=3158161.3158170>
- [126] T. B. Sousa, A. Aguiar, H. S. Ferreira, and F. F. Correia, “Engineering software for the cloud: Patterns and sequences,” in *Proceedings of the 11th Latin-American Conference on Pattern Languages of Programming*, ser. SugarLoafPloP '16. USA: The Hillside Group, 2016, pp. 16:1–16:8. [Online]. Available: <http://dl.acm.org/citation.cfm?id=3124362.3124381>
- [127] W. Dawoud, I. Takouna, and C. Meinel, “Elastic virtual machine for fine-grained cloud resource provisioning,” in *Global Trends in Computing and Communication Systems*, P. V. Krishna, M. R. Babu, and E. Ariwa, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 11–25.
- [128] A. Ullah, J. Li, Y. Shen, and A. Hussain, “A control theoretical view of cloud elasticity: taxonomy, survey and challenges,” *Cluster Computing*, May 2018. [Online]. Available: <https://doi.org/10.1007/s10586-018-2807-6>

- [129] David Hadka. (2018) A Free and Open Source Java Framework for Multiobjective Optimization. [Online]. Available: <https://github.com/MOEAFramework/MOEAFramework>
- [130] K. Deb and R. B. Agrawal, "Simulated Binary Crossover for Continuous Search Space," Indian Institute of Technology, Kanpur, India, Tech. Rep. IITK/ME/SMD-94027, Nov. 1994.
- [131] K. Deb and R. B. Agrawal, "Simulated Binary Crossover for Continuous Search Space," *Complex systems*, vol. 9, no. 2, pp. 115–148, 1995.
- [132] K. Deb and M. Goyal, "A combined genetic adaptive search (geneas) for engineering design," *Computer Science and informatics*, vol. 26, pp. 30–45, 1996.
- [133] I. Ari, B. Hong, E. L. Miller, S. A. Brandt, and D. D. E. Long, "Managing flash crowds on the internet," in *11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems, 2003. MASCOTS 2003.*, Oct 2003, pp. 246–249.
- [134] Google Developers , "Usage limits." [Online]. Available: <https://developers.google.com/sheets/api/limits>
- [135] M. Haklay and P. Weber, "Openstreetmap: User-generated street maps," *IEEE Pervasive Computing*, vol. 7, no. 4, pp. 12–18, Oct 2008.
- [136] M. F. M. Firdhous, D. L. Basnayake, K. H. L. Kodithuwakku, N. K. Hatthalla, N. W. Charlin, and P. M. R. I. K. Bandara, "Route advising in a dynamic environment – a high-tech approach," in *Innovations in Computing Sciences and Software Engineering*, T. Sobh and K. Elleithy, Eds. Dordrecht: Springer Netherlands, 2010, pp. 249–254.
- [137] S. van Vugt, "Setting up web services," in *Beginning Ubuntu LTS Server Administration*. Springer, 2008, pp. 325–349.
- [138] N. Matthew and R. Stones, *Beginning Databases with PostgreSQL*. Apress, 2005.
- [139] T. Preston-Werner. (2018) Semantic Versioning 2.0.0. [Online]. Available: <https://semver.org>

- [140] David Hadka. (2018) MOEA Framework: A Free and Open Source Java Framework for Multiobjective Optimization. [Online]. Available: <http://moeaframework.org>

# Acronyms

**AKS** Azure Kubernetes Service. [24](#), [49](#), [65](#), *Glossary:* [AKS](#)

**Amazon ECS** Amazon Elastic Container Service. [24](#), *Glossary:* [Amazon ECS](#)

**Amazon EKS** Amazon Elastic Container Service for Kubernetes. [24](#), *Glossary:* [Amazon EKS](#)

**AP** Application Provider. [1](#), [2](#), [4](#), [5](#), [7](#), [22](#), [24](#), [34](#), [73](#), [86](#), [101](#), [108](#), [129](#), *Glossary:* [AP](#)

**APP** Application placement problem. [38](#), [40](#), *Glossary:* [APP](#)

**BStrength** Bottleneck strength of a resource. [14](#), [15](#), [51](#), [56](#), [61](#), [84](#), [109](#), *Glossary:* [BStrength](#)

**CaaS** Container-as-a-Service. [8](#), [23](#), [24](#), [31](#), *Glossary:* [CaaS](#)

**CDI** Cross-over Distribution Index. [75](#), [77](#), *Glossary:* [CDI](#)

**CP** Cloud Provider. [2](#), [7](#), [22](#), [24](#), [34](#), *Glossary:* [CP](#)

**CSP** Constraint Satisfaction Problem. [63](#), [108](#), *Glossary:* [CSP](#)

**ESA** Exhaustive Search Algorithm. [6](#), [72](#), [76](#), [90](#), [97](#), [101](#), [109](#), *Glossary:* [ESA](#)

**GA** Genetic Algorithm. [5](#), [15](#), [75](#), [83](#), [84](#), *Glossary:* [GA](#)

**GCE** Google Compute Engine. [23](#), [164](#), *Glossary:* [GCE](#)

**GCP** Google Cloud Platform. [164](#), *Glossary:* [GCP](#)

**GKE** Google Kubernetes Engine. [24](#), [49](#), [65](#), [164](#), *Glossary:* [GKE](#)

**I** Iterations. [18](#), [75](#), [169](#), *Glossary:* [I](#)

**IaaS** Infrastructure-as-a-Service. [8](#), [23](#), [31](#), *Glossary:* [IaaS](#)

**LBM** Layered Bottleneck Mutation. [5](#), [80](#), [109](#), [130](#), *Glossary*: [LBM](#)

**LBX** Layered Bottleneck Crossover. [5](#), [78](#), [109](#), [130](#), *Glossary*: [LBX](#)

**LQN** Layered Queueing Network (LQN). [4](#), [5](#), [11](#), [35](#), *Glossary*: [lqng](#)

**MDI** Mutation Distribution Index. [75](#), [77](#), *Glossary*: [MDI](#)

**P** population size. [75](#), *Glossary*: [P](#)

**PaaS** Platform-as-a-Service. [8](#), [31](#), *Glossary*: [PaaS](#)

**PGA** Performance-oriented Genetic Algorithm. [5](#), [6](#), [30](#), [37](#), [72](#), [75–78](#), [80](#), [84](#), [85](#), [90](#), [97](#), [98](#), [101](#),  
[107](#), [109](#), [110](#), [165](#), *Glossary*: [PGA](#)

**PM** Polynomial Mutation. [77](#), [81–83](#), [98](#), [167](#), *Glossary*: [PM](#)

**RSA** Random Search Algorithm. [6](#), [72](#), [75](#), [90](#), [97](#), [101](#), [107](#), [109](#), *Glossary*: [RSA](#)

**SaaS** Software-as-a-Service. [8](#), [31](#), *Glossary*: [SaaS](#)

**SBX** Simulated Binary Crossover. [77–79](#), [98](#), [167](#), *Glossary*: [SBX](#)

**SGA** Simple Genetic Algorithm. [16](#), [77](#), [168](#), *Glossary*: [SGA](#)

**SGAP** Simple Genetic Algorithm for Performance. [6](#), [72](#), [75](#), [77](#), [78](#), [90](#), [97](#), [101](#), [107](#), [109](#), [167](#),  
*Glossary*: [SGAP](#)

**SPE** Software Performance Engineering. [9](#), *Glossary*: [SPE](#)

**SUT** System under test. *Glossary*: [SUT](#)

# Glossary

**AKS** Azure Kubernetes Service is a Container-as-a-Service offering provided by Microsoft. [24](#), [170](#)

**Amazon EC2** Infrastructure-as-a-Service provided by Amazon Web Services for creating and managing VMs. [23](#)

**Amazon ECS** Amazon Elastic Container Service. [24](#), [170](#)

**Amazon EKS** Amazon Elastic Container Service for Kubernetes. [24](#), [170](#)

**AP** Application providers, are the customers of the Cloud providers and provide software for use by the end-users. They have access to the services provided by the Cloud providers but are unaware of the underlying cloud platform. The role of the Application provider is also context-specific. If a Cloud provider consumes the services of another Cloud provider and then provides a service to the end-user, then in this particular context, the former has assumed the role of the Application provider. In this thesis, the discussion involves those Application providers who have control over the provisioning decisions of their application tasks onto VMs and container instances. [1](#), [170](#)

**APP** Application placement problem refers to the decision making involved in placing application within VMs or containers such that the application can meet given objectives. [38](#), [170](#)

**application architecture** A model describing the application tasks and their interactions. The deployment configuration and application architecture together describe a deployed application. [3](#), [26](#), [35](#), [70](#), [72](#), [73](#)

**application-cloud-model** A library implemented in Java, that allows for simplified representation of applications, container types and clouds. application-cloud-model library is available on Maven Central through its artifactId `ca.appsimulations.models`. [116](#)

**autoscaling** Autoscaling is another term for [dynamic provisioning](#), where resources are automatically scaled up and down, based on a given autoscaling policy, due to changes or disturbances in the environment. A provisioning system maintains the state of the system, adhering to the specified policy.. [2](#), [28](#), [163](#)

**BStrength** Bottleneck strength of a resource. [14](#), [170](#)

**CaaS** Container-as-a-Service is a service offering provided to the AP. Offered services include a containerized environment.. [8](#), [170](#)

**CDI** Cross-over Distribution Index is a parameter of Simulated Binary Crossover operator. The parameter determines how near the offsprings are to their parents, with larger values creating more nearness. [75](#), [170](#)

**CI/CD pipeline** Continuous Integration and Continuous Delivery/Deployment pipeline is an automated software solution to integrate the work being done on a software artifact on a frequent basis through code merges and testing and move the artifact from one environment to another based on tests (unit, integration, end-to-end, acceptance and smoke). A concrete example would be development of an editor which involves multiple software developers. The work of each developer would take place in their own branch and then sent out for review to their peers through pull-requests. The pull-requests would trigger builds on the pipeline software (e.g. Bamboo). The builds will compile and run tests on the editor software. Once the software has passed testing and is approved by the peers, it can be merged to the main “master” branch. This is the continuous integration functionality of the pipeline. Once the code is merged it most, a new build will be triggered on the pipeline which would build the artifact and publish it to a central repository, such as Artifactory. The artifact would then be automatically deployed to a development environment. After going through tests, the software will further be deployed to the staging environment. At this point, the artifact can be manually or automatically deployed — after tests — to production, where it can be available to the end-users. If the deployment happens manually — say through an approval process — then this is the continuous delivery functionality of the pipeline. Automatic deployment to production after the tests is the continuous deployment functionality of the pipeline. [72](#), [89](#)

**CP** Cloud providers host services through their infrastructure and platform for use by the application providers. [2](#), [170](#)

**crossover** Crossover operation reproduces the selected individuals to create offsprings. Here, two parents are chosen for producing two children until a new generation of size  $P$  is generated. The process works by picking a random crossover point and then mixing parts of the parent strings to form the children. e.g. for parents: 11110 and 10001, with a randomly picked crossover point being after the first two digits, the resulting children would be 11001 and 10110. [17](#)

**CSP** Constraint Satisfaction Problem is a problem where the goal is to satisfy the given constraints. [63](#), [170](#)

**decision maker** In dynamic provisioning, the decision-maker decides the amount of scaling for an application such that the specified objectives of the application are met. Examples of such objectives may include but are not limited to performance, cost, energy consumption etc. . [2](#), [31](#), [70–73](#), [116](#)

**decision scheduler** In dynamic provisioning, the decision scheduler decides when the provisioning decision maker is invoked. [31](#), [53](#)

**deployment configuration** The type of containers and the number of containers for each service, form the deployment configuration. The deployment configuration, together with the application architecture, describe a deployed application. [4](#), [5](#), [31](#), [47](#), [53](#), [54](#), [65](#), [70–73](#)

**dynamic provisioning** Dynamic provisioning is another term for [autoscaling](#), where resources are automatically scaled up and down, based on a given autoscaling policy, due to changes or disturbances in the environment. A provisioning system maintains the state of the system, adhering to the specified policy.. [2](#), [162](#)

**elite** In elitism, each iteration of the genetic algorithm saves an elite (or best) individual, to add it back in to the next generation. The cost of the elite individual is the lowest. [75](#)



**encoding** Encoding is the mapping of the problem variables to the individual in the GA. e.g. one example of encoding is a binary number where each digit represents the problem variables. 16

**end-user** End-users are customers of Application providers. The latter develop and deploy their software for use by the former. Provisioning decisions made by application provider are for accommodating the workloads generated by the end-users. 2, 22

**entry** In LQN modeling, entries represent the operations of a task.. 12

**ESA** Exhaustive Search Algorithm (ESA) is an search algorithm introduced in this thesis to navigate through all the deployment configurations and find the optimal configuration that meets the given objectives and constraints. 6, 170

**evaluation** In genetic algorithm, the evaluation operation evaluates a population using a [fitness function](#). 17

**fitness function** In genetic algorithm, each population is evaluated using a fitness function. For an optimization problem, this is the objective function. 17, 164

**GA** Genetic Algorithm is a search algorithm, based on evolutionary theory, where the goal is to solve an optimization problem. 5, 170

**GCE** Google Compute Engine provides Infrastructure-as-a-Service to create and manage VMs on [Google Cloud Platform \(GCP\)](#). 23, 170

**GCP** Google Cloud Platform is a cloud platform provided by Google. Provided services such as [GCE](#) and [GKE](#). 164, 170

**gene** In genetic algorithm, a gene makes up an individual. e.g. an individual represented as 010, has 0, 1 and 0 as the genes. 16, 80, 83

**git** Git is distributing versioning control software. The git client is installed on the user's machine so that they can initialize new version controlled projects or pull in existing projects. The projects can reside locally on the user's machine or can be remotely hosted git servers. Example of remotely hosted git servers are <http://github.com> and <http://gitlabs.com>. 119

**GKE** Google Kubernetes Engine is a Container-as-a-Service offering provided by Google. [24](#), [170](#)

**I** In genetic algorithm, iterations are the number of generations of the initial populations that are created until the algorithm is terminated. This number also indicates the number of times the algorithm repeats the cycle of crossover and mutation on the initial population to generate each new generation. [18](#), [170](#)

**IaaS** Infrastructure-as-a-Service is a service offering provided by the Cloud provider. Offered services include VM instances, Storage, Networking, etc. . [8](#), [170](#)

**individual** In genetic algorithm, an individual is made up of genes. Individuals together form a population. In a binary encoding, an individual is represented by a binary string or an array, composed on digits zeros and ones. e.g. an individual 010011. [16](#)

**initialization** The genetic algorithm begins with the initialization operation, where a population of size  $P$  is randomly generated. Any other initializations required for the algorithm would also happen here. [16](#)

**jLQNInterface** A library implemented in Java, that provides APIs for building, solving, analyzing, and manipulating LQN models. jLQNInterface library is available on Maven Central through its artifactId `ca.appsimulations.jLQNInterface`. [51](#), [52](#), [116](#)

**layered bottlenecks** Layered bottlenecks are either hardware or software resources that have a saturation value above a given saturation threshold. For software bottlenecks, instead of simply relying on their saturation, a *BStrength* metric is also used. The task that is a layered bottleneck have the highest *BStrength* value . [15](#), [78](#), [80](#)

**LBM** Layered Bottleneck Mutation (LBM) is a bottleneck based mutation operator, proposed in this thesis, as part of the [PGA](#) algorithm. [5](#), [171](#)

**LBX** Layered Bottleneck Crossover (LBX) is a bottleneck based crossover operator, proposed in this thesis, as part of the [PGA](#) algorithm. [5](#), [171](#)

**lqng** Layered Queueing Networks (LQN) analytical models are based on extended QNs and are designed to eliminate the shortcomings of Queueing Networks (QN). They are well-suited to

depict both complex software applications and the hardware resources that these software entities run on [18]. LQNs are ideal for representing the interactions and intricacies of multi-tier applications.. 4, 171

**maven project** A project that uses the Apache's Maven tool for building, packaging and/or publishing the project. These details are which are specified in a [pom](#) file. The [pom](#) file also uniquely identifies the project using groupId and projectId. Further details about using the Maven tool is available at <https://maven.apache.org/index.html>. 117, 167

**MDI** Mutation Distribution Index is a parameter of Polynomial Mutation operator. The parameter determines how near the offspring is to their parent, with larger values creating more nearness. 75, 171

**Microsoft Azure IaaS** Infrastructure-as-a-Service provided by Microsoft Azure. 23

**MOEA** A open-source Java framework that allows developing and experimenting with multiobjective evolutionary algorithms (MOEAs). <http://moeaframework.org>. 116

**monitoring agent** Monitoring agent is a service that monitors the state of the cloud system and save the information in a monitoring database. The information made available by the monitoring agent is useful for analysis, various decision making and for alerting the operators of any issues in the system. 72

**mutation** In genetic algorithm, mutation operation follows the crossover operation. Mutation changes the genes of an individual. e.g. in a binary encoding, a parent represented as 010 could be mutated to 011 through mutation of the last gene. Here, one parent after mutation produces a child. 17

**P** In genetic algorithm, population size is the number of individuals that are created in each generation. 75, 171

**PaaS** Platform-as-a-Service is a service offering provided by the Cloud provider to allow further development through the platform. Offered services include APIs, container instances, runtime environments etc. . 8, 171

**PGA** Performance-oriented Genetic Algorithm is a genetic algorithm that I have proposed in this thesis. This algorithm is based on [SGAP](#), where the focus on meeting performance constraints along with minimizing cost. However, Performance-oriented Genetic Algorithm differs from [SGAP](#) where instead of using [SBX](#) and [PM](#) operators directly, the former algorithm uses crossover and mutation operators that are based on bottleneck theory. [5](#), [171](#)

**PM** Polynomial Mutation operator simulates a binary mutation for real values when running a genetic algorithm. [77](#), [171](#)

**pom** A POM file is an XML file that includes specifies details of building, packaging and/or publishing a [maven project](#). Further details about the POM file is available at <https://maven.apache.org/guides/introduction/introduction-to-the-pom.html>. [117](#), [120](#), [124](#), [166](#)

**population** In genetic algorithm, a population represents the individuals in each generation. [16](#)

**proactive provisioning** Proactive provisioning is a dynamic provisioning approach which involves prediction based approaches to provision adequate resources before the changes in the workload is seen . [67](#), [72](#)

**probability of crossover** The crossover operation relies on the parameter: probability of crossover ( $p_c$ ). This parameter determines whether crossover will happen. If the crossover does not happen then the parents are copied directly as new population. [17](#)

**probability of mutation** The mutation operation relies on the parameter: probability of mutation ( $p_m$ ). This parameter determines whether mutation of a gene of the individual with take place. [17](#)

**processor** In LQN modeling, processors are entities on which a task runs on, where the processor uses time. Each processor has its own queue.. [12](#)

**provisioning agent** A service that adjusts resources on the cloud platform by adding or removing the resources based on configuration sent by the decision maker. [31](#), [70](#)

**reactive provisioning** Reactive provisioning is a dynamic provisioning approach which involves making provisioning decisions after workload changes are seen . [67](#), [72](#)

**response time** Response time is the total roundtrip time for a request to complete processing through the system. 9

**RSA** Random Search Algorithm (RSA) is an population based random search algorithm. 6, 171

**SaaS** Software-as-a-Service is a service offering provided to the end-user. Offered services include email services, web applications, etc. When providing this service offering, the entity takes on the role of an Application provider.. 8, 171

**SatisfyQoS** SatisfyQoS is a provisioning algorithm that uses layered bottlenecks for decision making. The goal of the algorithm is to satisfy response time constraints. This algorithm is my contribution as mentioned in this thesis. The algorithm has been published in [15] . 3, 5, 30, 37, 47, 54, 61, 84, 108, 110

**SBX** Simulated Binary Crossover operator simulates a single-point binary crossover for real values when running a genetic algorithm. 77, 171

**selection** In genetic algorithm, the selection operation involves choosing individuals based on their fitness for reproduction. The higher the fitness value of an individual, the higher the probability that the individual will be chosen for reproduction. 17

**SGA** Simple Genetic Algorithm is a genetic algorithm, originally proposed by John Holland. 16, 171

**SGAP** Simple Genetic Algorithm for Performance is a genetic algorithm that I have proposed in this thesis. This algorithm is based on SGA with focus on meeting performance constraints along with minimizing cost. 6, 171

**SPE** Software Performance Engineering is a well-defined systematic process where attention to the system's performance is paid from an early software development stage and followed throughout the software development lifecycle. 9, 171

**target system** In a control feedback loop, the target system is the system that is being controlled. With regards to the provisioning decision maker that is presented in this thesis, the target system is the application running in the cloud that is managed by the Application Provider.

From a performance perspective, the idea is to meet response time constraints and minimize cost by managing the application scaling. 71

**task** Tasks are resources in LQN modeling. These resources may be software processes, customers or hardware devices, etc. Each task has a queue to receive requests. Tasks execute on a processor.. 12

**termination criteria** A genetic algorithm continues selection, crossover and mutation operations to create new generations until the termination criteria is satisfied. An example of the termination criteria is I count, which is the number of generations of the initial population. 18

**throughput** Throughput is the rate of customer request completion. 9

# Index

- Crossover, [17](#)
- Encoding, [16](#)
- Mutation, [17](#)
- Response time, [9](#)
- Selection, [17](#)
- Throughput, [9](#)
  
- Amazon EC2, [23](#)
- Amazon Elastic Container Service (ECS), [24](#)
- Amazon Elastic Container Service for Kubernetes (EKS), [24](#)
- application, [24](#)
- application architecture, [3](#)
- Application Placement Problem (APP), [38](#)
- Application Provider (AP), [1](#)
- application-cloud-model, [116](#)
- AppPlace, [26](#)
- AppScale, [26](#)
- autoscaling, [2](#)
- Azure Kubernetes Engine (AKS), [24](#)
  
- BStrength (bottleneck strength), [14](#)
  
- CI/CD pipeline, [72](#)
- Cloud Provider (CP), [2](#)
- Cloud provisioning, [21](#)
- Constraint Satisfaction Problem (CSP), [63](#)
  
- Container-as-a-Service (CaaS), [8](#)
- Cross-over Distribution Index (CDI), [75](#)
  
- decision maker, [2](#)
- decision scheduler, [31](#)
- deployment configuration, [4](#)
- dynamic provisioning, [2](#)
  
- elite, [75](#)
- end-users, [2](#)
- entries, [12](#)
- evaluation, [17](#)
- Exhaustive Search Algorithm (ESA), [6](#)
  
- fitness function, [17](#)
  
- gene, [16](#)
- Genetic Algorithm (GA), [5](#)
- git, [119](#)
- Google Cloud Platform (GCP), [164](#)
- Google Compute Engine (GCE), [23](#)
- Google Kubernetes Engine (GKE), [24](#)
  
- individuals, [16](#)
- Infrastructure-as-a-Service (IaaS), [8](#)
- initialization, [16](#)
- Iterations (I), [18](#)
  
- jLQNInterface, [51](#)

Layered Bottleneck Crossover (LBX), 5  
 Layered Bottleneck Mutation (LBM), 5  
 layered bottlenecks, 15  
 Layered Queueing Network (LQN), 4  
  
 many-to-many, 29  
 many-to-one, 29  
 maven projects, 117  
 Microsoft Azure IaaS, 23  
 MOEA, 116  
 monitoring agent, 72  
 Mutation Distribution Index (MDI), 75  
  
 one-to-many, 29  
 one-to-one, 29  
  
 Performance-oriented Genetic Algorithm (PGA),  
     5  
 Platform-as-a-Service (PaaS), 8  
 Polynomial Mutation (PM), 77  
 pom, 117  
 population, 16  
 Population size (P), 75  
 proactive provisioning, 67  
 probability of crossover ( $p_c$ ), 17  
 probability of mutation ( $p_m$ ), 17  
 processor, 12  
 provisioning agent, 31  
  
 Random Search Algorithm (RSA), 6  
 reactive provisioning, 67  
 ResScale, 26  
  
 SatisfyQoS, 3  
 service, 25  
 Simple Genetic Algorithm (SGA), 16  
 Simple Genetic Algorithm for Performance (SGAP),  
     6  
 Simulated Binary Crossover (SBX), 77  
 Software Performance Engineering (SPE), 9  
 Software-as-a-Service (SaaS), 8  
 steps, 21  
  
 target system, 71  
 tasks, 12, 24  
 termination criteria, 18  
 tier, 24  
 types, 21  
 VmPlace, 26  
 VmScale, 26