UNIVERSITY OF TARTU
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
Institute of Computer Science
Software Engineering

Alireza Ostovar

Optimal Resource Provisioning for Workflows in Cloud

Master's thesis (30 ECTS)

Supervisor: Satish Narayana Srirama, Phd

TARTU, 2014

Optimal Resource Provisioning for Workflows in Cloud

Abstract

Cloud computing has gained significant popularity over past few years. Employing service-oriented architecture and resource virtualization technology, cloud provides the highest level of scalability for enterprise applications with variant load. This feature of cloud is the main attraction for migration of workflows to the cloud. Since each task of a workflow requires different processing power to perform its operation, at time of load variation it must scale in a manner fulfilling its specific requirements the most. Scaling can be done manually, provided that the load change periods are deterministic, or automatically, when there are unpredicted load spikes and slopes in the workload. A number of auto-scaling policies have been proposed so far. Some of these methods try to predict next incoming loads, while others tend to react to the incoming load at its arrival time and change the resource setup based on the real load rate rather than predicted one. However, in both methods there is need for an optimal resource provisioning policy that determines how many servers must be added to or removed from the system in order to fulfill the load while minimizing the cost. Current methods in this field take into account several of related parameters such as incoming workload, CPU usage of servers, network bandwidth, response time, processing power and cost of the servers. Nevertheless, none of them incorporates the life duration of a running server, the metric that can contribute to finding the most optimal policy. This parameter finds importance when the scaling algorithm tries to optimize the cost with employing a spectrum of various instance types featuring different processing powers and costs. In this paper, we will propose a generic LP(linear programming) model that takes into account all major factors involved in scaling including periodic cost, configuration cost and processing power of each instance type, instance count limit of clouds, and also life duration of each instance with customizable level of precision, and outputs an optimal combination of possible instance types suiting each task of a workflow the most. We created a simulation tool based on the proposed model and used 24-hour workload of ClarkNet ISP to conduct performance experiments. The results of experiments suggest that our optimal policy can minimize the cost of running a workflow in the cloud.

Keywords: Cloud computing, resource provisioning policy, workflow, task, instance, instance type, cost, workload , LP model, optimal model.

Ressursside optimaalne varustamine tvoogudele pilves

Lhikokkuvte

Pilvearvutuse populaarsus on viimaste aastate jooksul mrkmisvrselt kasvanud. Kasutades teenustele orienteeritud arhitektuure ning virtualiseerimist, vimaldab pilv varieeruva koormusega skaleerumist eelkige ettevtetele suunatud programmidele. See on ks suuremaid phjuseid miks tvoogusid pilve migreeritakse. Kuna iga tvoo osa vajab vastavalt kas rohkem vi vhem ressursse, siis pilve poolt pakutud ressurssid peavad skaleeruma nii, et see htiks tvoo vajadustega. Resursside skaleerimist saab teha manuaalselt, eeldades et tkoormuse muutusperioodid on deterministlikud, vi automaatselt, kui tvoos esineb ettearvamatuid koormuse tuse ning langusi. Seni on esitatud mitmeid automaatse skaleerumise ideid. Mned neist meetoditest proovivad ennustada, kui palju koormust vib esineda, samal ajal kui teised meetodid proovivad ressursse pakkuda alles koormuse kohale judmise ajal.Mlema meetodi puhul leidub aga vajadus strateegia jrgi, mis tagaks, et ressursse varustataks optimaalselt ehk tuvastada, kui mitu serverit tuleb lisada vi eemaldada ssteemist, et rahuldada koormuse nudlus ning samal ajal minimiseerida ka kulu. Antud magistrits esitatakse lineaaprogrammeerimisel phinev meetod, mis arvestab peamisi tegureid skaleerimises nagu, kulu, konfiguratsiooni hind, masinate judlus, pilve mahtuvus ning ka iga tmasina kestvus. Antud andmete phjal tagastatakse optimaalne kombinatsioon vimalikest instantsitpidest mis rahuldaks igat tvoo alamosa kige paremini. Lisaks loodi ka simulatsioon antud mudeli testimiseks ning katsete jooksutamiseks. Tulemuste kohaselt on nha, et pakutud meetod vhendab tvoogude jooksutamise hinda pilves.

Mrksnad: Pilvearvutus, resursside tagamise poliis, tvoog, t, instants, hind, judlus, LP mudel, optimaalne mudel.

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank my parents for helping me throughout the life with their endless support and kindness. I would like to acknowledge prof. Marlon Dumas and my nice supervisor prof. Satish Srirama for all of their efforts and attentions for me during these fantastic years of studying in University of Tartu. And at the end I would like to thank University of Tartu for giving me the chance of being a part of this old prestigious university.

Chapter 1

Introduction

1.1 Introduction

A workflow is composed of a set of activities utilizing computer systems as computational resources to achieve a particular goal. Breaking a complex experiment into small components, workflows can simplify execution and analysis of a heavy computation. Each component runs a small piece of the main process and the resulted output will be the input of next one and this chain of components complete the initial large job. Analysis and reexperimenting these small tasks are much easier and faster. Furthermore, the tasks that process high volume of data or need high processing power can run on strong remote servers, and the ones that require less resource usage can be deployed locally. Nowadays, creating web services hosted on the cloud is the most popular approach to implement and present each of these activities. Data analysis, simulation and image processing can be counted as some common ways of using workflows.

Cloud computing is basically science of using communication networks, specifically Internet, for connecting a large number of computer systems to provide a quality service to the user. This service can be software(SAAS[1]), infrastructure(IAAS[2]) or platform(PAAS[3]). This phenomenon has several advantages. First of all the resources can be allocated to the applications on demand and can scale up and down dynamically, meaning that you pay more

[1]Software as a service
[2]Infrastructure as a service
[3]Platform as a service

just when you need to. Second advantage to this model is that companies can avoid high infrastructure costs in the beginning of their business and therefore focus more on their products rather than hosting. Since maintenance and high availability of resources is the responsibility of cloud provider, companies can save a lot of time and money on this burdensome task. The numerous servers supplied by clouds provide a great environment for deployment of any applications including workflows. But why is this useful to workflows or why should we move workflows to cloud?

Even though taking the burden of maintenance and offering the pricing policy of 'pay as you go' are remarkable benefits of cloud computing, resource virtualization and service-oriented architecture can be deemed as the main attraction of it. Virtualization is the art of converting rigid physical infrastructure into soft flexible resources that can be supplied to the user using web services. Cloud computing adopts the concept of service-oriented architecture to provide its functionality as easily accessible services, facilitating resource provisioning and instance invocation in the cloud. Furthermore, virtualization empowers cloud providers to offer resources with different hardware power. Maximizing hardware utilities, virtualization also enables dynamic resource allocation, which is the base of cloud scalability, what makes cloud so interesting for workflows.

As mentioned above, workflow applications are composed of tasks, some of which are highly resource-intensive and some just perform light computations. The popular mechanism for running each task is binding them a cluster of instances managed by a head server which distributes the requests among them (1). It is possible to allocate a fixed number of servers to each cluster, provided that the quantity of load is known and also it is invariant. However, this is not the case in most workflow applications, since the number of computations or amount of data to be processed fluctuates significantly most of the time. This necessitates use of dynamic resources allocation of cloud computing, provided through virtualization and its service-oriented architecture. This way, workflows can employ more resources at time of load increase and return them when there is no need for them anymore. This dual capability of cloud is called elasticity and plays the primary role in migrating workflows to cloud.

1.1.1 Motivation

Load variation of applications follow different models, depending on the service they provide. As an example a shopping website experiences the workload peaks on holidays, requiring higher processing power, while it can handle all incoming requests with few servers on working days. In contrast there are applications such as Wikipedia which have high demand throughout the year. In case of known load change, the resource allocation can be performed manually, with adding or removing static number of instances, but all applications incur some unpredicted load fluctuation, necessitating an automatic mechanism for supplying servers. Auto-scaling is a feature of cloud computing helping to add and remove instances on demand and transparent to the user. In contrast to manual scaling that user must extend or shrink the size of an instance cluster using commands or graphical interface of cloud provider's system, auto-scaling takes care of all functions required for invoking and terminating instances at time of request rate variation. The only action done by user is configuring an auto-scaling plan matching the workload of their application. As one of the most useful features of cloud, auto-scaling is highly suitable for applications that experience hourly, weekly or monthly variation in their workload.

Auto-scaling services monitor the status of instances inside the same group and based on defined parameters decide on change in number of running instances. The efficiency of an auto-scaling mechanism mainly depends on its resource allocation policy, evaluated by request loss rate and total resource cost. So the big challenge here is trying to maximize throughput while minimizing the cost. Some methods address this problem through forecasting future loads and supplying resources beforehand, and some choose to react to the incoming load after their arrival, being more cautious in resource provisioning. There are some advantages and disadvantages to both methods. While in predictive methods request loss rate is usually lower, it will end in higher cost. There are also approaches that tackle this challenge using a combination of predictive and reactive models.

In a workflow, tasks run in parallel or sequence and perform unique operations. These operations require different processing power and take different amounts of time to run, so at time of load increase while addition of another instance can solve problem of a task, there might be need for extending cluster of another task by several instances. For example, in an image processing flow,

the task designated for compression requires lower processing power than the task for object detection, so they must be scaled in manners specific to them. Moreover, in every cloud there are a range of instance types, each one having a different power/price rate. Whereas a large instance might be more beneficial for one task, a medium might have a better performance for another one, so we shall allocate different instance types to different tasks in order to reduce the cost. However, sometimes the performance gap between these instance types is not that wide and the optimal solution resides in a composition of them.

Hourly charging is a popular pricing policy among prominent cloud vendors such as Amazon. Based on this policy that is an implementation of 'pay as you go' model, cloud provider charges the cost of the whole hour once the instance enters another hour of its life, disregarding if it will fill that hour or it will live just for a fraction of it. This makes the decision on employing a new instances or extending a running one more crucial, since we have to pay for the whole hour even if we just need it for a couple of minutes. Moreover, in case we incorporate various instance types in our setup, we have to compare the price per request rate of each pair of them and depending on life duration of each instance, choose the most optimal setup. For example, if there are two small instance running at the 55th minute of their paid life, and a medium instance can handle the same workload as sum of both of them with lower price, maybe we can minimize the cost through starting a new medium instance instead of extending current running small instances for another hour.

The common approach adopted by application providers for improving Quality of Service(QOS) as well as reducing cost is hosting their application in multiple clouds. A recent research (2) has revealed that the decent average waiting time of loading a page is 2 seconds or less, no matter the user is using a mobile device or connecting through personal computer. The demand for a service might vary in each geographical location, so it is more beneficial to place each service in a cloud located as close as possible to the region with highest request rate, resulting in lower response time for most clients. Additionally, clouds have different performances in doing different tasks, for example while a cloud handles more requests which require more processing power than data transfer, another cloud might be more efficient in requests involving massive data transfer. Therefore the optimal approach is locating each service in a cloud suiting it the most.

As a result, the goal is finding a method which incorporate all major factors involved in scaling of a system, and will offer the most optimal resource setup of each task in a workflow.

1.1.2 Contributions

As we concluded in previous section, we need a complete method that receives parameters related to system scaling, and outputs the most optimal setup. In this paper we will achieve this method using an LP model. In this model each service can be located in a separate cloud possessing different policies and infrastructure. Major inputs to our model include incoming workload of each task, processing power, periodic cost and configuration time of each available instance type in the cloud hosting the cluster of the task, maximum instance count limit of the cloud corresponding to each task, and age of each running instance. Feeding the LP model with mentioned inputs, it outputs the optimal number of instances from each type that must be added to or removed from cluster of each task, resulting in handling the workload and minimizing the cost. This novel optimal LP model contributes to the resource provisioning systems to choose optimal setup of instances at any point of time of running the servers. Based on this model we will create a restful web service which will be accessible through a HTTP request.

1.1.3 Outline

In next chapter we will elaborate on problem that have been addressed by this thesis. In third chapter we will look at the state of the art in field of auto-scaling mechanisms. Chapter 4 is dedicated to the description of tools and technologies used for implementation of the idea. Then, in chapter 5 we will explain method description and implementation in details. Experimental results will be outlined in chapter 6. Finally we will conclude the paper in chapter 7 and will give some ideas for future work in chapter 8.

Chapter 2

Problem Statement

To fulfill needs of a bigger range of customers, cloud providers offer a vast range of instance types, from Micro type to 2Xlarge, each of which has different processing power, memory, storage and network bandwidth (3). There are also optimized instances that target applications with specific needs, requiring more CPU power, for example image processing applications, higher memory for tasks such as data analysis, or wider network bandwidth for more communicative systems. The price of each type varies proportionally to its capabilities. In addition, these instances are located in different regions, which further affects their prices. For example, Amazon provides its infrastructure in five different regions including Asia, Europe, South America, US East and US West. For instance, while a m1.small instance in US East costs $0.044 per hour, the same instance costs $0.058 per hour in Asia Pacific. Therefore, we need a model that finds the optimal number of instances from each instance type in each region that minimizes the cost, and still handles the workload.

However, a model that takes into account all of mentioned factors and provides us with the best arrangement of servers is not still fully optimal, since it does not incorporate lifetime of current running instances. As mentioned above, customer is charged on an hourly base for each instance, and since each instance is added to the system on demand and hence at different times, at some point of time there might be several different-type instances that have lived for different amounts of time. In order to find the optimal solution, age of an instance must also be taken into account. Let us explain the problem with an example. Let us assume that we have two instance types of small and medium available for our application. After bombarding each of them and

carrying out several experiments we found out the the small type can handle 6 requests per second and the medium one 12 r/s. The price of small type is \$0.25 per hour and the medium costs \$0.4 per hour. We assume that current workload is 6 r/s and therefore we have one small instance running. Suppose that after 50 minutes the workload increases to 12 requests per second. If we do not consider age of instances the best solution is to add another small instance, however, this is not the optimal solution. If we add another small instance, assuming that 12 r/s workload persists, after ten minutes we must pay for the first instance as well. As a result we will have spent \$0.5 for the two small instances after ten minutes(ignoring the first payment for the initial small instance). Now let us add a medium instance instead of the small one at time of load change(after 50 minutes). In this case, when the first instance fills its paid hour it will be shut down, since our medium instance can handle 12 r/s and we do not need the initial small one anymore. As a result we have paid less (\$0.4) and we still satisfy the workload. The only issue left is that 10-minute time that first small instance can still live, which must be involved in the calculations. In first scenario, since we take advantage of this instance for 10 minutes, we must subtract this profit from the calculated cost, and in the second scenario this amount of money is an additional cost, because we can already handle the incoming load using the added medium instance and these last ten minutes the application is overprovisioned with servers. The 10-minute cost of a small instance is \$0.04, which is calculated by dividing hourly price of small instances by sixty minutes and then multiplying by 10. A summary of all stated calculations is listed here:

$$
\begin{aligned}
&Cost\ of\ first\ scenario\ =\ (cost\ of\ two\ small\ instances)\ - \\
&(10 - min\ profit\ of\ a\ small\ instance) = 0.5 - 0.04 = 0.46 \\
&Cost\ of\ second\ scenario\ =\ (cost\ of\ a\ medium\ instance)\ + \\
&(10 - min\ cost\ of\ a\ small\ instance) = 0.4 + 0.04 = 0.44 \\
&Saved\ cost\ by\ employing\ second\ scenario\ instead\ of\ first\ one\ = \\
&(Cost\ of\ first\ scenario)\ -\ (Cost\ of\ second\ scenario) \\
&= 0.460.44 = 0.02
\end{aligned}
\tag{2.1}
$$

So we will save \$0.02 by adding a medium instance instead of a small one.

Therefore, we need a model that, in addition to above-mentioned factors, takes account of life duration of current instances and outputs the most optimal

setup of instances. In this paper we will propose a generic LP model which takes into account all of these factors and produces the most optimal setup of resources, minimizing cost and fulfilling workload. Then we will create a restful web service based on our model which can be utilized by real-life workflows as the resource provisioning policy of their scalability management system. In our model, the capacity of each task in workflow can scales up and down at customizable intervals independently and based on its own current load, resulting in prevention of possible bottlenecks in the whole flow. We will create a simulation tool for benchmarking our model through typical workflow control structures, Parallel and Exclusive. We will conduct the same experiments using the optimal method that does not consider life duration of running instances wt time of setup estimation and will compare the results.

In the following chapter, we will look at related works that have been conducted so far in order to minimize request lost and resource cost in the realm of Auto-scaling. Then we will explain how our method differentiates from them, and how it can contribute to them.

Chapter 3

Related Work

As said before, Elasticity is the main attraction of cloud computing for enterprise applications. However finding a proper method of scaling is not an easy task. Depending on the reason of load change, estimating number of resources for handling the workload can be performed statically or dynamically (4). For example for most applications there are some seasonal load change which is predictable and therefore can be fulfilled through adding excessive servers beforehand. But every typical application has some load spikes which are not foreseeable and pose losing requests. That is where dynamic automatic scaling emerges, to solve the problem of unpredicted varying load without involvement of application provider.

The common pattern in deployment of workflows to the cloud is designating a cluster of virtual instances to each of its tasks. In this model, each cluster has a master node coordinating slave nodes which actually do the real job. MapReduce (5) (6) is a popular programming model that adopts master/slave approach. The master node disseminates the load among its slave nodes and hence it must also keep track of work distribution. The optimization in this model can be enforced on the distribution part such that nodes receive proportional work to their power. Kepler (7) is a tool that takes advantage of MapReduce model in order to facilitate execution of workflows. In our model also each task has its own cluster and scales independently.

Scaling servers can be done horizontally, by adding new servers or vertically, by improving existing servers through for example enhancing CPU power or adding additional RAMs. However, most operating system do not support changing physical components without rebooting the system, hence most cloud

providers just offer horizontal scaling. Scaling can cause two problems; it can cause over-provisioning, meaning that there are more instances than needed, this can be caused by for example taking peak loads as our base to scale up. Opposed to that, it might lead to under-provisioning, which happens when current resources cannot handle the load, which can be triggered by considering average load as the new load.

Auto-scaling policies can be classified into two groups of reactive methods and proactive methods (8) (9). Reactive ones decide on resource provisioning based on last values derived from monitoring tools. These methods can have lower performance, due to their delay for adding servers, resulting from late discovering of load change or the time duration needed for adding a new instance which can take several minutes sometimes. This is where proactive methods might appear to be more useful. However, these methods also have some disadvantages such as resource overprovisioning or underprovisioning, caused by imprecise prediction.

Each auto-scaling mechanism is composed of two policies, one for detecting when to scale and one for determining how to scale. The former one specifies if the system must be extended or shrunk and also the requirements that must be fulfilled by system. The later one, however, tries to fulfill the requirements set by previous policy by finding the optimal amount of resources that must be added to or eliminated from the system. There are many algorithms presented for finding the suitable setup of instances, targeting throughput maximization and cost minimization. In below we will discuss some of the policies that have been introduced for resource provisioning so far.

3.1 Static Threshold-based Policies

Having been used by many auto-scaling services such as Amazon Auto-Scale (10), Scalr (11) or RightScale (12), threshold-based policies are so popular among users due to their simplicity. In this model there are two rules, one for scale-up and the other for scale-down. Each rule can include several performance metrics such as CPU usage, request rate, response time, network traffic and etc., which can be specified by user. Each of these metrics includes multiple parameters for setting upper or lower thresholds, the quantity type(avg, maximum, minimum,) and duration of defined state. User can also deter-

mine minimum and maximum number of instances and the amount of extra instances that must be added or removed at each time of scaling. At runtime when the conditions fulfill the defined requirements auto-scaling service alarm is triggered and automatic scaling is performed. There is another parameter for the time gap between scalings, called cool-down time, that can also be defined by user and turns off the service for the specified timespan. Proper setting of these parameters vary among applications according to their workload, and there is need for expert knowledge of load and cloud computing to set up an optimal service. This is one of the major disadvantages of this method.

Typically, just one of the mentioned metrics is set for scaling, such as the work done by Dutreilh et al. (13) or Han et al. (14), where response time is used as performance metric. However, Hasan et al. (15) claimed that a combination of these metrics must be incorporated in order to achieve more optimal setup. RightScale is a cloud management system that helps applications to run on a distributed platform using multiple clouds. It facilitates taking advantage of multiple clouds with configuring them through graphical user interface and without distinguishing among underlying infrastructure of each of them. This system has a enhanced auto-scaling mechanism that incorporates all running instances in scaling, through a democratic voting process, such that scaling will happen if a user-specified percentage of instances vote in favor of it (16). Even though this method has been adopted by several projects (17) (18), it still has the drawback of dependency on user-defined threshold value. Simmons et al. (19) introduce a strategy-tree to bypass this problem. In this method three customized policies are defined and the algorithm tries to find the one matching the workload the most and switches between them.

3.2 Reinforcement Learning

Reinforcement Learning (20) is a machine learning algorithm that can be used for auto-scaling purpose. This method has a decision-maker component, called agent, which based on current state of the environment outputs the suitable action which should be taken next, gradually leading to a table containing the map of state-actions pairs. In addition to current state, previous selected action will also affect decision of the algorithm to choose the next action. So the tricky part of this method is finding a proper policy to find next optimal action.

Even though this method is not originally an auto-scaling approach, but it can fit into an auto-scaling problem (21), for example number of current virtual instances can be defined as state, changing request rate as environment and the action is finding optimal number of instances fulfilling the load within a decent response time. This method has some disadvantages: since it is dependent on the experience and learning it does not have good initial performance, and the time it takes to converge to an optimal policy can be quite long. Moreover, performance of this approach is suitable if the load incurs smooth changes, while if there are sudden bursts in load it cannot react well. An improvement to this model was proposed by Xavier Dutreilh et al. (21), contributed to better initialization and faster convergence to optimal policy.

3.3 Queuing Theory

Queuing theory (22) is mathematical representation of waiting lines. This model can be used in business processes when size of requests inside the queue and their waiting times matter. This method can also be applied to auto-scaling, where we try to scale resources in order to reduce response time. A queue is described as A/B/C, where A represents inter-arrival time distribution, B is service-time distribution and C describes number of running servers. We consider a queue for each of tasks in workflow; combination of these queues form a queuing network. Given workload rate, the goal is to find optimal amount of resources that can serve maximum requests in queue within an acceptable response time; this can be achieved by minimizing waiting time inside the queue and also service time of servers. Capacity of queue can be defined with a fixed number or as infinite, meaning that no received requests will be rejected. Bhuvan Urgaonkar et al. (23) experimented a network of queues in a multi-tier application. He derived future workload from a load predictor and based on that found the adequate number of servers that can handle the load within the desired response time.

3.4 Control Theory

Control theory (24) is an engineering model composed of a controller and the system. The controller employs suitable logic to produce an optimal output

based on the dynamic input and feedback of previous output to the system. This model has been widely used in context of auto-scaling. There are different types of controller component, but the most efficient ones are able to adjust themselves to the load and the feedback of system at runtime, some advanced ones are even able to reconfigure themselves. The controller can accept one or multiple inputs (e.g. workload rate and/or target response time) and produces one or multiple outputs (e.g. number of servers). Inside a controller there is a function, called transfer function, that maps the inputs to the proper outputs. A linear programming model can be used in this component, however, the most common methods are kalman filter and fuzzy logic. Harold et al. (25) propose a simple controller that produces the output based on average CPU usage. Ali-Eldin et al. (26) in another work suggest to consolidate an adaptive and proactive controller for scaling-down process and a reactive model for scaling-up. Fuzzy controllers are also well-used, in which workload as input is mapped to the optimal amount of resources as output. Xu et al. (27) utilized a fuzzy model to estimate CPU capacity needed for handling the incoming workload.

In a framework built by Martti Vasar et al. (28), they combined a heuristic model with a reactive model for auto-scaling of simulated MediaWiki application. In his model they assumed that servers start at the same time and therefore, 5 minutes before the end of each hour, they estimated the required amount of resources based on the average workload of current hour and trend of loads in past few hours. Even though they achieved proper results in their experiments, the assumption of method that all servers start at the same time does not conform to real-life application. Just imagine that one server malfunctions and hence becomes replaced by a new server, this new server will have a new timeline different from others. Another disadvantage of this method is its long interval (one hour) between two epochs of resource provision, weakening this method against sudden load spikes.

In current resource provisioning methods just one instance type that matches the task the most is used for the setup of servers. Bakabs (29) is a mobile application for managing load of web applications in cloud. Carlos Paniagua et al. in this work propose a novel LP model that can find optimal number of resources from each instance type in each cloud that can fulfill workload of every task of the system hosted in a different region in the same or different

cloud, while minimizing the cost. Our model will take a similar approach, with some extensions to solve the problem outlined in 1.1. and will output the most optimal setup of resources at any given point of time. In order to find the most optimal configuration, all possible setups must be checked. This process must be undertaken in a decent amount of time, meeting the scaling speed required by the application. We will delegate this task to a linear programming solver that takes various inputs such as processing power and cost of various instance types in different clouds and also life duration of each running instance and outputs the most cost-optimal configuration of servers, fulfilling the incoming workload. None of the resource provisioning policies so far has taken lifetime of an instance into account in order to optimize the cost. Our model calculates the cost of killing and retaining an instance at any point of time and decides on removing or keeping it. It also find the cost-effective instance replacements that might reduce the cost further.

Chapter 4

Background

In order to implement the method we have utilized several tools and techniques that will be outlined briefly in this section.

4.1 Linear Programming (LP)

Linear programming (30) is a method to minimize or maximize a target variable calculated by a linear function, subject to linear equality and inequality constraints. As one of the mathematical optimization models, linear programming is usually used in problems targeting cost or profit optimization. Intersects between the inequality constraints build a bounded region, so-called feasible region, which contains the possible values of the involved variables. The point inside this region(if exists) that minimizes or maximizes, depending on the goal, the linear objective function is the optimal solution. The general form of a linear programming problem is as follows:

A linear objective function to be optimized:

$$f(x1, x2, x3,) = c1x1 + c2x2 + c3x3 + ... \tag{4.1}$$

Inequality constraints:

$$\begin{aligned} &a11x1 + a12x2 + a13x3 + ... <= b1 \\ &a21x1 + a22x2 + a23x3 + ... <= b2 \\ &... \end{aligned} \tag{4.2}$$

Equality constraints:

$$\begin{aligned} & e11x1 + e12x2 + e13x3 + ... = d1 \\ & ... \end{aligned} \tag{4.3}$$

Non-negative variables:

$$x1 >= 0,\ x2 >= 0,\ x3 >= 0,\ ... \tag{4.4}$$

4.2 OptimJ

OptimJ (31) is a Java-based modeling language for solving optimization problems including linear programming, mixed integer programming and nonlinear programming. This utility is an extension of Java programming language and is supported by the popular programming environment, Eclipse. Due to Java-based nature of OptimJ, developers can have access to the whole Java library inside OptimJ modules. OptimJ has an easy interface to define decision variables, linear objective function and constraints, using straightforward keywords and structures. The engine of OptimJ translates the code, written by developer using the provided interface, to pure Java code at compile time, calling all required optimization functions. The OptimJ code is written inside a file having .optimj extension; at compile time a file with the same name but with .java extension, containing calls to optimization methods, is generated out of this file. This tool has several LP solvers such as GLPK (32), lpsolve (33), CPLEX (34) and MOSEC (35) that can be used for solving LP problems; first two ones are free and open source and are used for more simple problems and the two last ones are commercial and are utilized for complex problems. Since our model is pure LP and also we did not want our model to be dependent on commercial tools We have used GLPK solver in order to find the optimal solution for our optimization problem. GLPK solver has much better performance than lpsolve; this was discovered after testing both solvers in various test scenarios.

4.3 Virtual Server

As opposed to dedicated server that reserves the whole computer, a virtual server shares the resources of a computer with other virtual servers. Instead of

dedicating a computer to each server which is a costly strategy, by dividing the resources of a high-power computer between many virtual instances, we can offer a wide variety of servers with different computing features to customers with various requirements at affordable prices, and still achieve almost all functionalities we could by using dedicated servers. There are many benefits for server virtualization; eliminating server sprawl, more efficient use of server resources, high server availability, facilitating test and development and low maintenance are among the main ones (36). However, if a virtual instance starts hogging resources, it can affect performance of other servers. These days the number of virtual servers deployed is much higher than the number of physical servers. Virtual servers play the main role in popularity of cloud computing, allowing application providers to consume as much resources as they need as well as save cost and time for maintenance practices.

4.4 Load Balancing

Load balancing is act of distributing the workload across multiple computing resources, e.g. back-end servers, in order to maximize throughput and minimize response time. Load balancing can be achieved through using hardware such as a multilayer switch(MLS), software such as a domain name system(DNS) or a combination of both. Eliminating dependency of the application on one server through load dissemination across multiple servers will result in higher reliability for the whole system. Sometimes servers are dispersed across multiple regions so that in case of a break in one region, other regions can still be responsive, reducing downtime of the system. One of the most prevalent use of load balancing is providing an Internet service, such as a popular website, from multiple servers, known as server farm. For Internet services, load balancer is usually a software program that distributes requests among several back-end servers, making end user see the whole system as one server replying th requests. Adding another tier in front of vulnerable back-end servers will result in better security, gained through hiding back-end servers and the internal structure of their network from users.

There are several scheduling algorithms that have been used by load balancers in order to choose the server to send the request to. These can range from simple methods such as random or round robin to the more complex al-

gorithms considering factors such as recent response times, number of active connections or capabilities of instances.

Large distributed systems might use multi-tier architecture including several load balancers in front-end tiers. In addition to hardware load balancers, there are many software solutions such as Zen LB (37) and Nginx (38). An important issue incurred by load balancing is persistence of user sessions across multiple requests so that the result of a resource-consuming operation, that must be done for any new session, can be stored and fetched for the same user session. There are different techniques for solving this problem. A simple solution would be sending requests coming from the same user to the same machine. This requires storage of the session id and the IP of that machine in load balancer, known as stickiness. Even though this method might be fast, but in case the load balancer fails for any reason, all session information inside it will be lost. A common way to ensure session persistence is saving session information inside an in-memory session database, so-called memcached, which is shared between all back-end servers.

4.4.1 Nginx

Nginx is an open source web application accelerator that has been used by busiest websites such as Facebook and Hulu, helping them to be more responsive, scalable, fast and secure. This web server, that is being used by over 140 million websites (38), is capable of performing concurrency processing, URL switching, HTTP load balancing, SSL termination and caching for web applications. Nginx was written by Russian programmer Igor Sysoev (39) in C language, and uses network application layer for processing and redirecting requests. As opposed to popular Apache Tomcat web server that consumes a thread per connection, Nginx uses a limited number of processes, called workers, each of which is capable of handling thousands of requests per seconds. Nginx uses an event-driven model utilizing native operating system functions, what makes it a suitable load balancer for high-performance applications, while consuming very low CPU power. A commercial version of this web server, called Nginx Plus (40), has recently been released, possessing enterprise-class features such as health check, activity monitoring and on-the-fly reconfiguration.

Configuring Nginx as a load balancer is relatively easy, however, maximizing its power to handle high traffic load requires expert knowledge of server

configuration. Depending on your sake of using a web server, Nginx provides you with various modules. In order to perform load balancing using Nginx, it must be provided with addresses of back-end servers, this is achieved by using ngx_http_upstream_module. The back-end servers must be specified in this module using their IP or DNS addresses, which can have different ports listening on TCP and UNIX-domain sockets. The load balancing algorithm inside Nginx is weighted round robin, meaning that each server in upstream module has a weight (defaults to 1) and it will receive corresponding number of requests to its weight, for example if the weight of a server is 7 and another one is 1, the former will receive 7 requests and then the latter will receive one. If Nginx encounters an error when redirecting the request to a server, it will try next server until it finishes trying each of servers once, then the returned result to client will be the response received from the last server. It is also possible to specify the maximum number of failures using max_fails parameter (defaults to 1)that put a server out of consideration of Nginx for a time duration that can be determined using fail_timeout parameter (defaults to 10 second).

Another handy module of Nginx that can be utilized for finding the number of requests handled by Nginx server is HttpStubStatusModule. This particularly comes useful to the users needing to discover the input load to their application using request-per-second metric. We will also use this module in order to find load rate of servers in our experiments. We fetch the number of requests at predefined intervals and divide the difference by interval duration. This module also outputs the number of open connections as well as total number of accepted and handled connections. Another useful module of Nginx worth mentioning is ngx_http_log_module, which is designed for logging tasks.

PHP is a server-side scripting language that is used mostly for web development. This language has been used in more than 240 million websites (41) and has been installed in over 2.1 million web servers. PHP code can be interpreted by a web server having a PHP processor module. PHP applications have been widely deployed to Apache web server during past years. However, Nginx as a new web application server can be used to host PHP applications by means of FastCGI module, resulting in a faster application with higher throughput (42), benefiting from inherent speed of Nginx which is in turn

due to its asynchronous event-driven request processing mechanism. In our experiments we will run PHP scripts using FastCGI Process Manager(FPM) on top of Nginx web server.

One of the main features of Nginx required for a load balancer is its capability of reloading without losing any of current requests inside it, making it suitable for Auto-scaling mechanisms, needing load balancer to get reloaded after addition or elimination of servers while responding to the current requests. This is achieved using a simple method; first it renames the .pid file, containing the process id of Nginx master, to .oldbin, then it initiates new master and also worker processes. At this point two instances of Nginx are running, one handling old requests and the other new requests. After serving old requests, the old master process and its workers are gracefully shut down and request handling is done just by new master process and its workers. If Nginx cannot succeed to apply new configuration, it continues handling loads using old configuration.

4.5 Amazon EC2

Amazon Elastic Compute Cloud(EC2) (43) is a web service, helping to ease use of cloud re-sizable resources for developers. Amazon EC2 provides a simple robust interface for administration of cloud resources for the consumers, facilitating scalability management of applications through scaling-up and down just with one command at time of load change. This service has many benefits that we will outline them shortly here.

Elasticity: Amazon EC2 service empowers you to augment or shrink the capacity of your application within a couple of minutes. Since this is provided through a web service, user can commission as many servers as he wants just using a single command, making scalability of the application as simple and fast as possible.

Flexible Management Services: You can interact with your instances directly and through simple commands, for example you can start or stop them at any time, without losing any data on the data storage. Through use of server virtualization, explained above, Amazon EC2 provides users with different types of hardware or software platforms. For instance depending on your application's goal you have choice of selecting among a wide spectrum of in-

stance types, from general-purpose instances to compute-optimized, memory-optimized, storage-optimized or even instances with high performance in GPU operations. Size of Amazon instances starts from micro and increases to small, medium, large, xlarge, 2xlarge, 4xlarge and even 8xlarge. It should be noted that each instance type has this size range and for example 2xlarge in compute-optimized is different from its peer in memory-optimized. Additionally, consumers can choose between various operating system platforms including numerous Linux distributions and Windows servers.

Reliability: Due to robust network infrastructure and data centers, Amazon's instances ensure high level of reliability, but in case of failure each instance can be quickly superseded by a healthy one. There is also functionality of health check alarm that will, in case of health check failure, notify the user about the issue, resulting in lower downtime.

Security: Through use of Amazon Virtual Private Cloud(Amazon VPC) (44), you can isolate your instances from the public and have maximum security control over them. Amazon VPC lets you create a virtual private network of servers, which depending on your purpose, can be exposed to public via Internet or can be just accessible in a private network. You can also connect your local servers to the cloud instances using industry-standard encrypted IPsec VPN connections.

Amazon also provides SDKs (45) for popular languages in order to facilitate interacting with its instances as much as possible from inside your application. Currently libraries are prepared for languages: Java, .Net, PHP, Python, Ruby, Node.js, Android and IOS. The provided libraries can sit next to your application and modularize instance management of your application, as opposed to traditional approach for dealing with EC2 web service from the application via SSH bash commands. Using SSH bash commands from inside the code was not only a non-standard slow manner, but it was also very error-prone and hard to maintain, caused by discrepant structures of commands and high number of switches of each command that must be set. In contrast, well-documented Amazon SDKs have easy-to-use neat structures that, supporting all possible operations of Amazon EC2, they significantly enhance performance of cloud resource management systems.

4.5.1 Amazon EC2 IP Addresses

When an instance is launched it automatically receives two IP addresses, namely private and public IPs (46). The private one is not accessible through Internet and is used for creating an Amazon Virtual Private Cloud(VPC), which is an isolated network of instances taking advantage of amazon elastic cloud. The public IP address is for external connections through Internet, and can be accessible on the defined ports by user. Public IPs also allow the cloud user to establish an ssh connection to their instances and interact with them as they do with their personal computers in front of them. Public and private IP addresses will remain the same if the instance is rebooted, however, if the instance is stopped and restarted they both will receive new addresses. Another disadvantage of them is that these IPs are static, meaning that they cannot be changed once they are assigned at launch time. However, Most applications are used by millions of users and cannot change their IP address in case of any incident incurred by servers. Therefore Amazon offers Elastic IPs which can be assigned to the user instead of the instance. Elastic IP addresses can be bought and retained by the user until they decide to release them. These IPs can be associated to or disassociated from an instance at any time after its launch, and in case of a server crash, making the whole incident and the recovery transparent to the end user.

4.6 Amazon Regions and Availability Zones

Amazon EC2 resources are hosted in multiple independent locations throughout the world. Each location is comprising of a region and an availability zone (47). Regions are in separate geographical areas and contain isolated availability zones. Regions are totally independent of each other and connection between them can only be achieved through public Internet. This provides maximum fault tolerance ans stability desired by application providers. Even though very rare, but failures might occur in cloud instances and the risk of service unavailability can be avoided through hosting the application across multiple regions. Availability zones, however, are placed inside regions and can be isolated or in contact through low-latency connections. At time of instance creation, use can let the cloud system to choose the least best availability zone for the instance based on the system health and the capacity of

zone, or they can select the desirable availability zone in case they want the instance to be close to or separate from other instances. Providing resources across multiple locations also allows application providers to host their servers as close as possible to their customers, reducing response time and attracting more customers.

4.7 Tsung

Tsung (48) is a distributed stress testing tool written in Erlang, a language for building fault-tolerant distributed applications. Tsung is suitable for performance benchmarking of various protocols including HTTP, MySql, LDAP, SOAP, XMPP, PostgreSQL and WebDAV, and it can be easily extended to other protocols. This open-source tool can simulate enormous number of clients sending requests to a server concurrently from different IP addresses, achieved through operating system's IP aliasing. Dynamic sessions in Tsung allow you to change request arrival rates, each of which lasts for a different duration, making a highly dynamic workload curve. Tsung has capability of user thinktime simulation which can further contribute to resembling a real-life test. At the end of each benchmark, Tsung provides you with useful statistics and charts related to network throughput, response times, successful and failed requests of the conducted test.

4.8 Hyperićs System Information Gatherer (SIGAR)

Each operating system has its own modules for providing information about system resources such as CPU or memory, making performance statistics collection a demanding task for developers. SIGAR (49) is a cross-platform API, helping to monitor various metrics of the system. Using SIGAR you have instant access to CPU, memory, network connections, uptime, logins, open files and many other metrics in all operating systems without caring about underlying platform-dependent commands. The core API of SIGAR is implemented in C language, but it also provides bindings for Java, C# and Perl.

In this chapter we went through the methods and tools that are used in this project. This will help to grasp the method implementation and experiments

better in next chapters. In next chapter, we will describe how we solved the problem stated at chapter 2 using a novel LP model.

Chapter 5

Method Description and Implementation

In previous chapters we introduced the basic idea behind auto-scaling and explained why it is needed in enterprise applications with varying workload. We argued that to fulfill the incoming load the auto-scaling mechanism must extend or shrink capacity of system through adding or removing additional servers. There are two general groups of auto-scaling methods, predictive methods and reactive methods. Both methods encompass a wide range of policies to predict or discover the incoming load curve in order to take highest advantage of available resource for fulfilling maximum load while optimizing the cost. Therefore, in both methods there is need for a second-phase policy to estimate the suitable setting of instances that meets all requirements set by scaling policy of auto-scaling mechanisms. The inputs to this policy can be incoming workload rate, processing data size, traversal data size, etc. In order to have an optimized policy we should take advantage of various available instance types featuring different processing power and prices. This policy must also incorporate several other factors such as instance count limit set by cloud provider and configuration cost of each instance type. At different points of time, based on need, we invoke new servers, extend existing ones and shut down some of them. Therefore, After several hours of application's working, we will have a collection of different-type instances, each of which has a different age. This is a factor that has not been included in any of resource provisioning policies, and can influence the resource cost to some extent.

In this chapter, we will introduce and describe an linear programming model that, taking into account all mentioned factors, provides the most optimal setting of resources for each task of a workflow system running on different regions of different clouds at any given point of time. Before elaborating on the model we need to make a few definitions that will be used for explaining it.

Region: As we mentioned in previous chapters, based on the operation that each a task in a workflow performs it might have different performances in different clouds, so the best practice is to host each task in the cloud suiting it the most. Therefore, in our model we consider a region with its own independent characteristics for each task, this region can be in a different cloud or the same cloud as other tasks. Each region will scale independently based on its own incoming load entering its load balancer. Each region can also have its own capacity of instances as a parameter which will be shown by CC. The set of regions is shown with R, where each region hosts a task of workflow. We will use r as region notation.

Instance Type: Since each cloud provides a wide spectrum of instance types with different resources and prices, it is more beneficial for the application to take advantage of multiple instance types. Therefore, each region in our model can include multiple instance types, each having its own processing power(P), price per period(C), capacity constraint(CCT), and configuration time(CT). Configuration time specifies the time span needed for an instance to initialize and switches to running status. During this time the instance is not usable, causing a further cost which must be considered in addition to the periodic cost of the instance. This additional cost is named configuration cost. The set of instance types of region r is shown with Tr notation. Instance type will be marked with t letter.

Time Bags: Instances are typically charged on a periodic basis, such that when an instance enters a new period of its life it will be charged for the whole period. Depending on how we divide this period, during the period the instance resides in several time intervals, till it totally fills the period. In our model, these time intervals are called time bags. The length of this period can be set as a parameter, and based on desired level of granularity, number of time bags can be different, however, by default we consider one hour for period length and 60 number of time bags for each instance type. The choice of this

number stems from 60 minutes of each hour. Time bags will help us to position each instance at any given point of time. Each time bag can contain several instances running at this point of time, and over the time these instances travel through all time bags till end of their hourly life. Time bags of each instance type have the same fixed price calculated by dividing price per period of an instance type by total number of time bags. The set of time bags of instance type t in region r is shown with TBr,t notation. We will show time bags with tb notation.

Killing Cost: The money we lose when we kill an instance before it fills its paid period is called killing cost. Killing cost is calculated by first subtracting number of the time bag containing the instance from the total number of time bags and then multiplying the result by price of a time bag of containing instance type. Killing cost will be marked with KC.

Retaining Cost: Retaining cost stands the opposite of killing cost, so it is calculated by multiplying number of the time bag containing the instance by price of each time bag. Basically this is the cost of the lived duration of the paid period of an instance. Retaining cost will be marked with RC.

5.1 Method Description

Linear programming is a mathematical model targeting problems that must find the optimal solution among several possible solutions. Each LP model is consist of a linear objective function that must be optimized, subject to a set of equality or inequality constraints. These constraints make a feasible region for the problem, while the algorithm must try to find a point in this region that maximize or minimize the objective function. There are a set of parameters that are adjusted before running the model and there are variables that are assigned different values by the model in order to optimize the objective function.
Parameters of our model are listed below:

-Cr,t: Cost of a time period of instance type t running in region r.

-$CTBr,t$: Cost of a time bag from instance type t running in region r. This cost is calculated by dividing the cost of a period of instance type t by total

number of time bags.

-CTr,t: Configuration time of instance type t running in region r. This value must be specified by time bag metric. For example in our experiments in next chapter, we consider first 3 time bags as configuration time.

-KCr,t,tb: Killing Cost of time bag tb from instance type t running in region r.

-RCr,t,tb: Retaining Cost of time bag tb from instance type t running in region r.

-Xr,t,tb: The number of instances in time bag tb from instance type t running in region r.

-Pr,t: Processing power of instance type t running in region r.

-$CCTr,t$: Capacity constraint (or instance count limit) of instance type t running in region r.

-Wr: Workload of region r. This is the current incoming workload to the system and must be provided by the same metric as Pr,t. meaning that if Pr,t is calculated by request per second, Wr,t must be provided as request per second too.

-CCr: Capacity constraint (or instance count limit) of region r. In different clouds this number is different, for example in Amazon, by default customer can launch up to 20 instances, and if more servers are needed, customer must make an application for it.
Our model has 2 variables:

-Nr,t: The number of new instances from instance type t running in region r that must be added to the system.

-Sr,t,tb: The number of instances of time bag tb from instance type t in region r that must be shut down.

The model will try to find out how many new instances of each type must be added and how many instances of each time bag from each instance type must be removed to minimize the resource provisioning cost in each of the regions.
Therefore the objective function is as follows:

$$\begin{aligned} Min\ (&\sum_{i=1}^{n}\sum_{j=1}^{m} N_{r_i,t_j} * C_{r_i,t_j} + N_{r_i,t_j} * (CT_{r_i,t_j} * CTB_{r_i,t_j}) + \\ &\sum_{i=1}^{n}\sum_{j=1}^{m}\sum_{k=1}^{q} S_{r_i,t_j,tb_k} * KC_{r_i,t_j,tb_k} + \\ &\sum_{i=1}^{n}\sum_{j=1}^{m}\sum_{k=1}^{q} (X_{r_i,t_j,tb_k} - S_{r_i,t_j,tb_k}) * RC_{r_i,t_j,tb_k}) \end{aligned} \tag{5.1}$$

The optimization model contains following constraints that must be fulfilled:

-The workload constraint $\forall\ regionsr \in R$:

$$\sum_{j=1}^{m}(N_{r,t_j} + (\sum_{k=1}^{q} X_{r,t_j,tb_k} - S_{r,t_j,tb_k})) * P_{r,t_j} \geq W_r \tag{5.2}$$

-The cloud capacity constraint $\forall\ regionsr \in R$:

$$\sum_{j=1}^{m}(N_{r,t_j} + (\sum_{k=1}^{q} X_{r,t_j,tb_k} - S_{r,t_j,tb_k})) \leq CC_r \tag{5.3}$$

-The instance type capacity constraint $\forall\ instancetypest \in Tr$:

$$N_{t_r} + (\sum_{k=1}^{q} X_{t_r,tb_k} - S_{t_r,tb_k}) \leq CCT_{t_r} \tag{5.4}$$

-Shutdown constraint $\forall\ timebagstb \in TBr, t$:

$$S_{tb_{r,t}} \leq X_{tb_{r,t}} \tag{5.5}$$

-And:

$$\begin{aligned} N_{r,t} &\geq 0 \\ S_{r,t} &\geq 0 \end{aligned} \tag{5.6}$$

The objective function comprises sum of all costs attached to changing the arrangement of resources at any point of time. This function, which in our case is cost function, sums cost of new instances and their configuration, killing cost

of each instance that must be shut down and retaining cost of each instance that will continue living. For each region the model outputs the number of new instances of each instance type that must be added and number of instances of each time bag from each instance type that must be terminated so that the cost becomes minimal and all constraints fulfilled. Adding two parameters of killing and retaining costs additionally optimizes the model, since it also covers the situations that adding new instances with different power/price rate can fulfill the load and also replace the old instances, minimizing the cost as much as possible. Using time bags allows us to calculate these new defined costs for each instance at any point of time. Killing cost basically means the loss that we sustain if we kill an instance before it fills its paid time period. But, in situations that adding a new instance instead of renewing current instances is more beneficial, retaining these old instances can cause extra cost which is avoided using retaining cost parameter. Each time that we remove an instance we add its killing cost to total cost and also subtract its retaining cost. So, these two new parameters actually specify how valuable a running instance is still for us, the more the instance lives in its current time period the higher retaining cost and the lower killing cost become, and hence, the less valuable the instance becomes. This also guarantees that at time of scale-down the instances from the last time bags will have higher chance to be shut down. However, adding a new instance is bound to a new configuration process that triggers redundant cost which might make the addition of the new instance unprofitable. This is avoided by adding configuration cost to objective function. Since configuration time for each instance type might be different, the model defines a new parameter for each instance type in each region. Adding killing and retaining cost enables the model to solve the problem stated in chapter 2 and finds the most optimal resource setup for all tasks of a workflow at once, considering status of all currently running instances.

The constraint 5.2 is defined to ensure that the new setup will fulfill the incoming workload in each region. Furthermore, the total number of instances in each region must not exceed its capacity; this is fulfilled using the constraint 5.3. The constraint 5.4 checks that the number of instances of each instance type in each region does not surpass its limit. And finally using constraint 5.5 we make sure that from each time bag the model does not shut down more instances than it contains.

5.2 Method Implementation

The model is implemented in OptimJ (31) and using one of its free solvers, called GLPK (32). The implementation of model using OptimJ interface is quite complicated, but in order to make the method usable for auto scaling mechanism, we created a simple interface for the model which can be presented as a restful web service. This web service is currently hosted in heroku and accessible using following interface:

-Method: *POST*
-URI: *optimalpolicy.heroku.com*
-Request Body Content Type: *XML, JSON*
-Response Body Content Type: *XML, JSON*
The request body must contain the list of regions containing their instance types, which in turn include their time bags with the number of instances inside. An example of request body content with all possible parameters is presented below:

```
<?xml version=”1.0” encoding=”UTF-8” ?>
<regions>
    <region>
        <name>USEast</name>
        <workload>130</workload¿
        <capacityofinstances>80</capacityofinstances¿
        <instancetypes>
            <instancetype>
                <type>small</type>
                <costperperiod>0.25</costperperiod>
                <processingpower>6</processingpower>
                <configurationtime>3</configurationtime>
                <capacityofinstances>40</capacityofinstances>
                <timebags>
                    <timebag>
                        <number>1</number>
```

```
                <instancecount>0</instancecount>
            </timebag>
            .
            .
            .
            <timebag>
                <number>50</number>
                <instancecount>1</instancecount>
            </timebag>
            .
            .
            .
            <timebag>
                <number>60</number>
                <instancecount>0</instancecount>
            </timebag>
        </timebags>
    </instancetype>
    <instancetype>
        <type>medium</type>
        <costperperiod>0.40</costperperiod>
        <processingpower>12</processingpower>
        <configurationtime>3</configurationtime>
        <capacityofinstances>40</capacityofinstances>
        <timebags>
            <timebag>
                <number>1</number>
                <instancecount>0</instancecount>
            </timebag>
            .
            .
            .
            <timebag>
                <number>60</number>
                <instancecount>0</instancecount>
```

```
                    </timebag>
                </timebags>
            </instancetype>
        </instancetypes>
    </region>
</regions>
```

The response to the request will contain the number of new instances from each instance type in each region that must be added and the number of instances from each time bag contained by each instance type in each region that must be shut down. An example of the response body is presented below:

```
<?xml version="1.0" encoding="UTF-8" ?>
<regions>
    <region>
        <name>USEast</name>
        <instancetypes>
            <instancetype>
                <type>small</type>
                <launch>0</launch>
                <timebags>
                    <timebag>
                        <number>1</number>
                        <shutdown>0</shutdown>
                    </timebag>
                    .
                    .
                    .
                    <timebag>
                        <number>50</number>
                        <shutdown>1</shutdown>
                    </timebag>
```

```
                .
                .
                .
                <timebag>
                    <number>60</number>
                    <shutdown>0</shutdown>
                </timebag>
            </timebags>
        </instancetype>
        <instancetype>
            <type>medium</type>
            <launch>1</launch>
            <timebags>
                <timebag>
                    <number>1</number>
                    <shutdown>0</shutdown>
                </timebag>
                .
                .
                .
                <timebag>
                    <number>60</number>
                    <shutdown>0</shutdown>
                </timebag>
            </timebags>
        </instancetype>
    </instancetypes>
  </region>
</regions>
```

In this chapter we introduced a new generic LP model capable of finding the most cost-optimal setup of instances, fulfilling the incoming workload. The new concept of time bags was presented and based on that we calculated killing and retaining cost of each instance. Taking into account killing, retaining and

configuration costs of each instance the method outputs the number of new instances from each instance type and number of old instances of each time bag from each instance type in each region that must be shut down. Moving instances through time bags over the time, we can obtain the most optimal arrangement of instances at any point of time. The model is presented to applications through a web service. In next chapter we will benchmark the model using real incoming load of an application over a 24-hour period. We will design two test case scenarios experimenting the model using two basic workflow management components, namely Parallel and Exclusive gates. We will compare our result with the result of our own optimal policy without considering time bags concept or in other words without considering lifetime of current instances at setup estimation time.

Chapter 6

Experiments

In order to experiment performance of our model, we have designed a few test case scenarios. We have chosen two of basic workflow control structures that are essence of many other complex structures and are used in any typical workflow, namely Parallel and Exclusive. We have compared our model with a benchmark optimal model that considers all factors as our model except the age of instances, So in this method there will not be any time bag, and hence killing and retaining cost. However, before elaborating on the performance experiments, we will show how the proposed model can solve the problem stated in chapter 2.

6.1 Cost Minimization Test

In order to examine if the model finds the most optimal setup of instances, we wrote a code that lists all possible transformations of current configuration in order to support new incoming workload. Applying concept of our LP model for cost calculation, we measure cost of each of these transformations.

In the scenario described in section 2, there are two instance types, Small and Medium. The former one is capable of handling 6 requests per second and costs \$0.25 per hour, and the later one has higher processing power and can handle 12 requests per second and costs \$0.40 per hour. At the time of resource provisioning, there is one small instance running at 50^{th} minute of its paid one-hour life. While this instance could fulfill the previous workload, 6 r/s, it is not able to handle the new incoming load which is 12 r/s. Therefore, we need to add another instance to the system in order to increase processing

power of the system and avoid losing requests. The fast solution would be adding another small instance and in total raising the power of the system to 12 r/s, which is required by current workload. But as we proved before, the cost-optimal solution would be adding a medium instance and shutting down the running small instance. In order to increase possible setups for handling workload, we added another instance type, Micro with processing power of 3 r/s and price of $ 0.15 per hour. We applied the created program, which implements cost calculation function of our optimal model, to this problem and acquired results stated in table 6.1.

Current Setup	New Setup	Cost
1 Small	4 Micros	0.672
1 Small	1 Small, 2 Micros	0.523
1 Small	2 Smalls	0.471
1 Small	1 Medium	0.462

Table 6.1: All possible transformations for the cost minimization test scenario. Measured cost is based on the optimal policy's cost function.

As the results suggest, the optimal policy's cost function succeeded in discovering the most cost-effective solution, using time bags and measuring killing and retaining costs. The measured costs are slightly different from the calculated costs in section 2, which is due to adding configuration cost to the list of considered costs. Even though this was a simple scenario, it was sufficient to prove the capability of the proposed model in minimizing the cost. However, in the next section we will benchmark our model using a real workload and will show how it can perform in real-life scenarios.

6.2 Performance Test

The performance experiments in this section are designed to measure cost-effectiveness of the model, its CPU and time consumption while looking for the optimal setup, and theoretical request loss. Since the ability of an auto-scaling mechanism in reducing the response time or load loss mainly depends on its first-phase policy for determining when and how much the system must scale, here we did not focus on acquiring real-life values for these two metrics. Since

we simulated the resource provisioning system instead of addition or removal of real-life instances in our experiments, we can only calculate theoretical request loss, which does not have a real-life value, but it is sufficient for comparison purposes.

6.2.1 Test Environment

Amazon cloud, as one of the most popular public clouds, have been used in many research projects as the test bed. Amazon provides a wide variety of instances in multiple regions, fulfilling a broad range of customers with different requirements. In addition to various hardware platforms, customer can choose among several operating systems such as Microsoft Windows, Red Hat Linux, SuSE Linux and Ubuntu, each in both 32 and 64 bit versions. Therefore, we also chose Amazon cloud in USEast region as our test environment infrastructure. In our experiments we used four different instance types, m1.small, m1.medium, m1.large and c3.large. The first three are used as available instance types for resource provisioning of workflow, and the last one is chosen as the platform hosting each component of the experiment process. While M1 instance types are designed for general purposes, C3 include compute-optimized instances, targeted at more CPU-intensive operations. The features of all available instance types can be found in Amazon website (50). A short list of instance types, used in our experiments, along with their specifications is presented in table 6.2.

Instance Type	vCPU	ECU	Memory (GiB)	Instance Storage (GB)	Linux/UNIX Usage
m1.small	1	1	1.7	1 x 160	$0.044 per Hour
m1.medium	1	2	3.75	1 x 410	$0.087 per Hour
m1.large	2	4	7.5	2 x 420	$0.175 per Hour
c3.large	2	7	3.75	2 x 16(SSD)	$0.105 per Hour

Table 6.2: Instance types used in experiments.

We used Linux 64-bit as operating system of back-end instances running each task of workflow and also instances running workflow control system. Ubuntu 64-bit was used on the instance running tsung, and Microsoft Windows 64-bit was employed for running the simulation program, implementing our LP model. Our benchmark model also runs on an instance having Microsoft Windows 64-bit as OS.

ClarkNet (51) was an Internet Service Provider(ISP) in United States between 1993 till 2003. The workload of this ISP between August 28, 1995 till September 4, 1995 is publicly available (52). We cut a 24-hour load from 00:00:00 till 23:59:59 of August 29^{th} for our experiments. We cut the load by minute and scale it up to reach a workload of over 600 requests per second at peak time. Tsung is used to bombard the system using the prepared load. Tsung has been widely used for stress testing of applications, and is capable of sending thousands of requests per second and is able to change the request rate according to the defined sessions by user. We changed the load according to the scaled load of ClarkNet on a minute basis.

6.2.2 Simulation Tool

We created a simulation tool that acts as a resource provisioning system, meaning that it fetches the load entering each task of the workflow, and feeds it to the model to calculate the amount of resources needed to handle the load. Based on the decision made by the resource provisioning policy, the application adds new instances or removes the running ones in a virtual way. In fact, there is no instance addition or removal in real life, but this change in number of instances happens just in the memory, which is enough for keeping track of instance's lifetime. At launch time, the instance receives the local time of the server as a timestamp with millisecond precision. This timestamp can be used at any moment for finding the number of periods an instance has lived and its time bag in current time period. Having the time bag number, we can easily calculate the killing and retaining cost of the instance.

Each task of the workflow has an Nginx load balancer which receives the load from the workflow management system(WMS) and passes it to its back-end server. Since we do not add a real instance at time of load change, we considered just one instance running a dummy PHP code for each task of the workflow behind its load balancer. For fetching the load from the load

balancers, the HttpStubStatusModule of Nginx is used, such that an HTTP request is sent to the address of each load balancer and the status of the load balancer, including total number of served requests is received. Subtracting previous registered requests number from this new number will give us the load change during the last interval.

We consider 60 time bags for each instance type, one per minute, and update the instance setup of the workflow once per minute, each time based on the request rate of last minute. Configuration time for all instances are set to 3 minutes and each instance needs 3 minutes to finish the termination process, meaning that if an instance is set to be killed, it must be killed at 57^{th} minute of its last life period. But, if an instance passes this minute in its current period, it will not be killed anymore and is considered as an extended instance which will live another time period, adding the cost of the new period to the total cost. We let each instance live till this minute even if it is set to be terminated before. At each setup update epoch, we set the status of all the instances such that they can have another chance to continue living, and then we run the resource provisioning policy. If based on the output of running the policy an instance must be terminated, we set its status accordingly, and if it is in its 57^{th} minute, we will shut down the instance. In addition, according to configuration time, an instance is not considered as running until it passes 3 minutes of its current life period first time it is launched. So when calculating the current load capacity of each region, we just count the instances with running status. However, when running the policy in next updates we consider all launched instances even they are not still in running status. The whole simulation process is shown in the figure 6.1.

All tasks of the workflow can have three instance types of m1.small, m1.medium and m1.large. Based on the incoming load and processing power/price rate of these instance types, the resource provisioning policy searches for the most optimal combination of them which minimizes the cost and handles the workload. Each task of a workflow performs a specific operation, consuming specific amount of resources and taking specific amount of time. We considered a workflow with three tasks for our experiments, each of which does a different job. First task runs the Huffman coding, second one performs a Selection sort and task 3 conducts a Merge sort on each incoming request to the task. The codes are written in PHP language and run on Nginx server using its

FastCGI module. We bombarded all m1.small, m1.medium and m1.large instances using Tsung and measured their power with the metric of requests per second(RPS). The results are shown in table 6.3.

Instance Type	Huffman Coding (RPS)	Selection Sort (RPS)	Merge Sort (RPS)
m1.small	7	9	10
m1.medium	13	16	18
m1.large	18	20	22

Table 6.3: Processing power of instance types, with the metric of requests per second(RPS).

We compared our model with an optimal policy, emulating all features of our model for finding the most optimal setup, except time bag concept and hence killing and retaining costs. At time of scale up, this model finds the most optimal number of instances from each instance type in each region that handles the load difference from last load, and at time of scale-down the method eliminates the instances that reduce the cost of running the system the most, considering processing power of the instance, its price and the time span it has lived in its last paid period. Therefore, at scale-down, the method removes a combination of the instances from different instance types that minimizes the cost to the highest extent. We also let an instance live till its killing time, in our experiments 57^{th} minute. This way we ensure that we benchmark our model against the most currently available optimal model, which we will refer to it as mini-optimal model henceforth.

In order to remove effect of cloud capacity limit on results of our experiments, we set the capacity of cloud and each instance type to 100 instances, so that models can launch as many instances as needed for handling the load.

6.2.3 Test Case Scenario 1, Exclusive Structure

In the first test case scenario, we considered a workflow management system consisting of an Exclusive OR gate(XOR). In an Exclusive gate, each time just one of the tasks connected to output of the gate is triggered. Each output branch of an Exclusive gate can have a weight which specifies how often each

branch is activated, such that the branch with higher weight receives more load. In our experiments the branch connected to task 1(in region 1) has the weight of 60 and the branch connected to task 2(in region 2) has the wight of 40, meaning that 60% of the load is passed to the region 1 and 40% of it to the region 2. After receiving the successful response from the selected region, the request is redirected to the task 3(in region 3), so region 3 receives the sum of loads processed by region 1 or 2. The whole structure of this experiment on the cloud is represented in the figure 6.2.

In order to ensure that both optimal and mini-optimal models receive the same load, we created a load fetcher service that fetches the load from load balancers every minute. This service runs on a separate instance, and feeds the simulators, running the models, with last load of each region.

In tables 6.4, 6.5, 6.6, and 6.7 we can see the result of the experiments for region 1, 2, 3 and sum of regions respectively, after running for 24 hours in the cloud.

measurement	Mini-optimal	Optimal
total requests	18,681,174	18,681,174
theoretical request loss	503,340	481,740
theoretical successful requests	97.305%	97.421%
total cost of instances	44.381$	44.132$

Table 6.4: Test case scenario 1. Resource provisioning experiments results in region 1.

measurement	Mini-optimal	Optimal
total requests	12,523,026	12,523,026
theoretical request loss	301,080	257,160
theoretical successful requests	97.595%	97.946%
total cost of instances	23.538$	23.444$

Table 6.5: Test case scenario 1. Resource provisioning experiments results in region 2.

The results suggest that the optimal policy could defeat the mini-optimal policy in all regions, and hence in the whole workflow. We could reduce the cost

measurement	Mini-optimal	Optimal
total requests	31,204,200	31,204,200
theoretical request loss	844,140	838,140
theoretical successful requests	97.294%	97.314%
total cost of instances	52.303$	51.735$

Table 6.6: Test case scenario 1. Resource provisioning experiments results in region 3.

measurement	Mini-optimal	Optimal
total requests	31,204,200	31,204,200
total cost of instances	120.222$	119.311$

Table 6.7: Test case scenario 1. Resource provisioning experiments results in the whole workflow(sum of the regions).

of running the workflow in each of the regions, in region 1 $0.249, in region 2 $0.094, in region 3 $0.568 and in total $0.911. To calculate theoretical request loss, each time that we measure load rate(r/s) of last time interval, we subtract processing power of the whole region in this interval from this number and then multiply by the time interval duration, measured in seconds. Of course this is different from request loss of real instances in real life, but it is enough for comparing our optimal model with the benchmark mini-optimal model. We can see that, in all regions, our model can even handle higher percentage of requests than mini-optimal model, while reducing the cost.

Figures 6.3, 6.4 and 6.5 represent the incoming load curve, scaling curve and also instance type usage curve of regions 1, 2 and 3 respectively.

As we can see, in all three regions, both optimal and mini-optimal models followed the incoming load very precisely. The registered incoming load curve and scaling curve display the competition between models to stick to the load and minimize the cost very well. The number of instances from each instance type launched in each region can also be observed in figures. The results suggest that, in all regions, optimal policy used more small instances and less medium instances than mini-optimal. This is due to the higher processing power/price rate of small instances than medium instances in all three regions.

However, the difference in rates of small and medium instances is not that big, and whenever beneficial the models launch medium instances. Both models try to find the most optimal combination of different-type instances at each setup update. But, the mini-optimal policy does not change the setup unless it is needed as a result of load decrease or increase, while optimal policy might change the setup even if based on the incoming load there is no need for it. This is because optimal policy searches for the replacements between old instances with new different-type instances, which are able to handle the load and at the same time reduce the cost. Therefore, after launching a medium instance for reducing the cost in a specific situation, optimal policy tends to replace this medium instance with small ones whenever suitable in order to take advantage of higher power of small instances in handling the load. This is why optimal policy used less medium instances than mini-optimal. None of the models launched large instances in any regions, which is due to the low power/price rate of this instance type in executing the designed codes.

Since running an LP model can be extremely resource-intensive, we used SIGAR (49) to measure CPU utilization of running our model each time it was searching for the optimal solution. The results suggest that, the CPU usage of the optimal model rose up to 50% in extreme situations, with the average of 3.35%. The model did not take more than 12 seconds to find the optimal solutions, while the average time consumption was 66 milliseconds. Figure 6.6 displays the charts of both CPU and time consumption of the optimal model during the 24 hours of experiment.

6.2.4 Test Case Scenario 2, Parallel Structure

As opposed to exclusive gate, parallel gate triggers both of its output branches, meaning that both tasks connected to the output of this gate will be invoked. Therefore on each request the workflow management system first runs both task 1(in region 1) and task 2(in region 2) and after receiving the successful response from both regions, the request is redirected to the task 3(in region 3). The whole structure of this experiment on the cloud is represented in the figure 6.7.

After running the experiment for 24 hours in the cloud, we obtained the results shown in tables 6.8, 6.9, 6.10, and 6.11, for each of the regions and the whole workflow.

measurement	Mini-optimal	Optimal
total requests	31,204,200	31,204,200
theoretical request loss	862,860	846180
theoretical successful requests	97.234%	97.288%
total cost of instances	73.097$	72.878$

Table 6.8: Test case scenario 2. Resource provisioning experiments results in region 1.

measurement	Mini-optimal	Optimal
total requests	31,204,200	31,204,200
theoretical request loss	850,380	829,500
theoretical successful requests	97.274%	97.341%
total cost of instances	57.944$	56.979$

Table 6.9: Test case scenario 2. Resource provisioning experiments results in region 2.

In this workflow also the optimal policy could overcome the mini-optimal policy in all regions and hence in the whole workflow. In region 1 we could save \$0.219, in region 2 \$0.965, in region 3 \$0.568 and in total \$1.752 in 24 hours running of the experiment. The theoretical request loss in optimal policy was also less than its value in mini-optimal in all regions, meaning that the optimal model could stick to the workflow better than its rival, resulting in losing less requests while minimizing the cost.

The figures 6.8, 6.9 and 6.10 illustrate the incoming load, scaling and instance type usage curves of each of the regions.

Similar to the results obtained in exclusive gate workflow, here also both models could stick to the load very well. As in the first test case scenario, the small instance was used more in optimal model than mini-optimal. The CPU usage of the model hit a peak of 50% in many situations, and the maximum time taken by LP model to find the optimal solution was less than 20 seconds. The average CPU usage was around 5.56 and the average time consumption was registered 138 milliseconds. The CPU and time consumption charts of the model for finding optimal solution is displayed in figure 6.11.

measurement	Mini-optimal	Optimal
total requests	31,204,200	31,204,200
theoretical request loss	844,140	838,140
theoretical successful requests	97.294%	97.314%
total cost of instances	52.303$	51.735$

Table 6.10: Test case scenario 2. Resource provisioning experiments results in region 3.

measurement	Mini-optimal	Optimal
total requests	31,204,200	31,204,200
total cost of instances	183.344$	181.592$

Table 6.11: Test case scenario 2. Resource provisioning experiments results in the whole workflow(sum of the regions).

In summary, in this chapter we reported the results of the experiments carried out in order to prove the accuracy and efficiency of our optimal model. We experimented our model through three different scenarios, one for showing the accuracy of the model in finding the optimal solution and the other two for benchmarking the performance of the model against the most currently available optimal model, which here we called mini-optimal. The results in accuracy test case suggested that the model is able to discover the most optimal setup of instances at any time. And using performance test cases we saw that the model could defeat mini-optimal model in all regions, reducing cost while having lower request loss. Even though the cost reduction achieved by optimal model was relatively low, we should consider that the experimented workflows were composed of just one simple gate with three tasks, while in real-life workflows, there are many tasks connected through several gates.

The time and CPU consumed by our model to find the optimal solution had a very high variance, while most of the times being at zero, it peaked at CPU usage of 50% and time usage of 20 seconds for the intensive loads with the peak of over 600 requests per second. Therefore we can conclude that our novel generic LP model could always find the most optimal configuration of instances composed of various instance types, in each region of the workflow

within a few seconds.

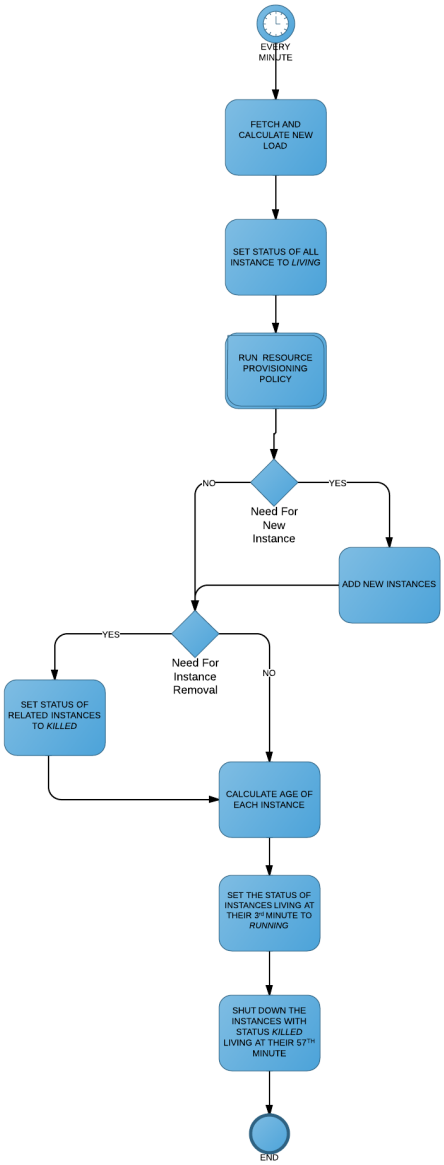


Figure 6.1: Simulation Process of Resource Provisioning System.

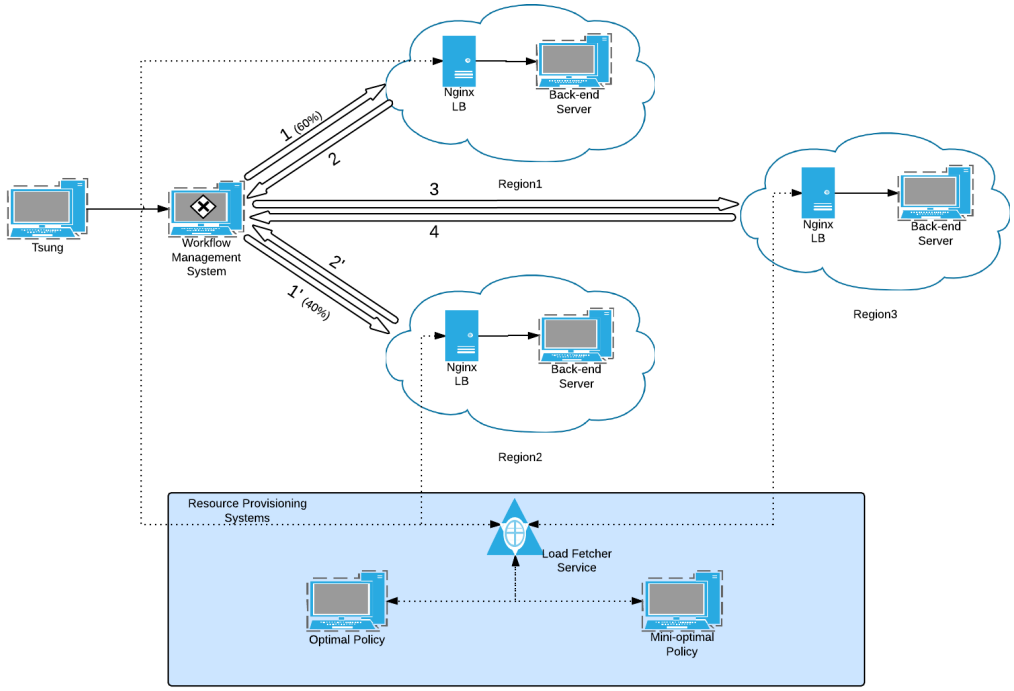
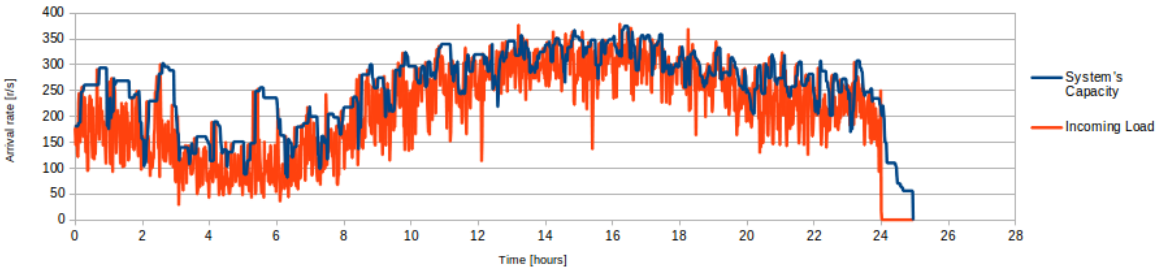
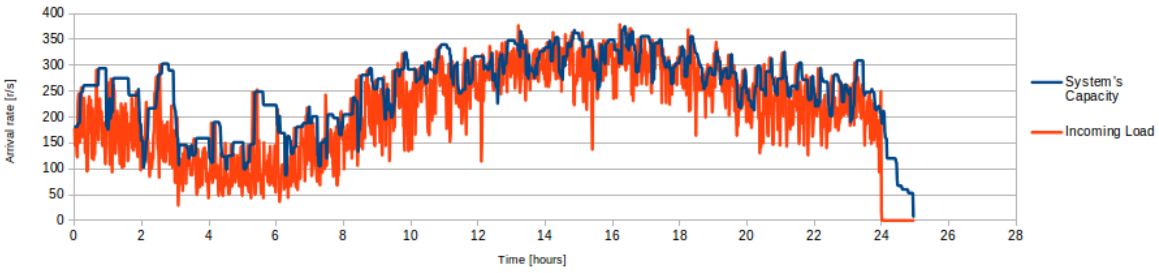


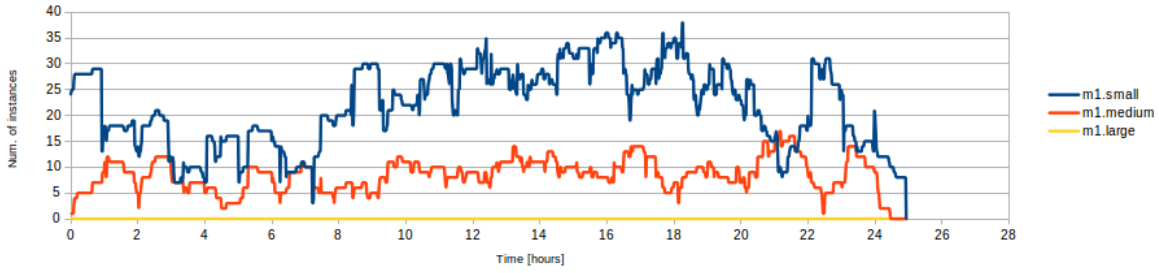
Figure 6.2: Test Case Scenario 1, with workflow management system, consisting of XOR gate.



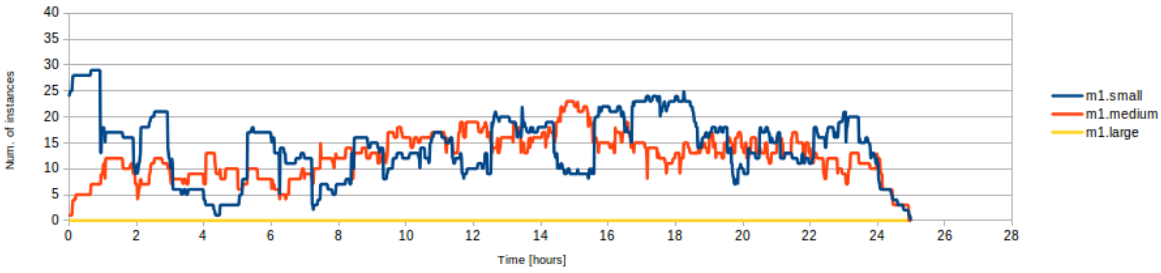
(a) Incoming load curve and scaling curve - Optimal



(b) Incoming load curve and scaling curve - Mini-optimal

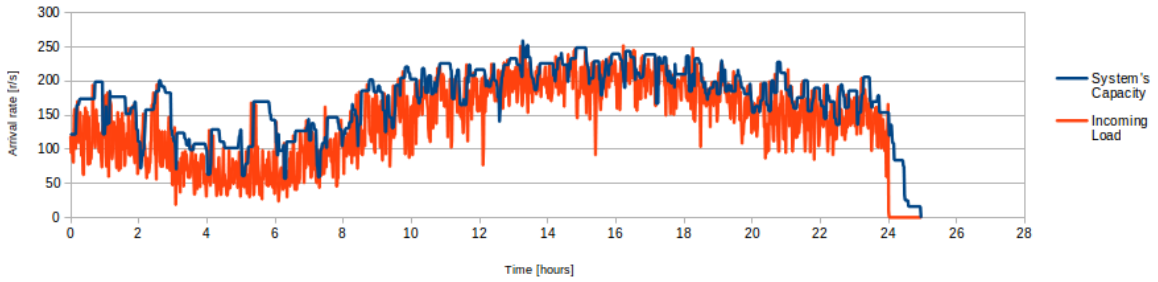


(c) Instance type usage curve - Optimal

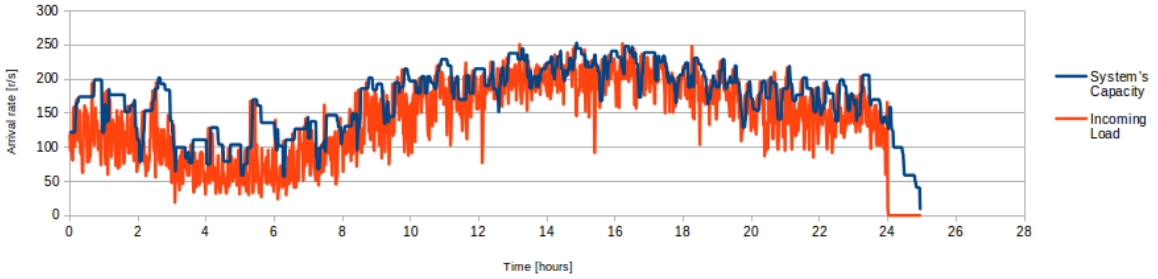


(d) Instance type usage curve - Mini-optimal

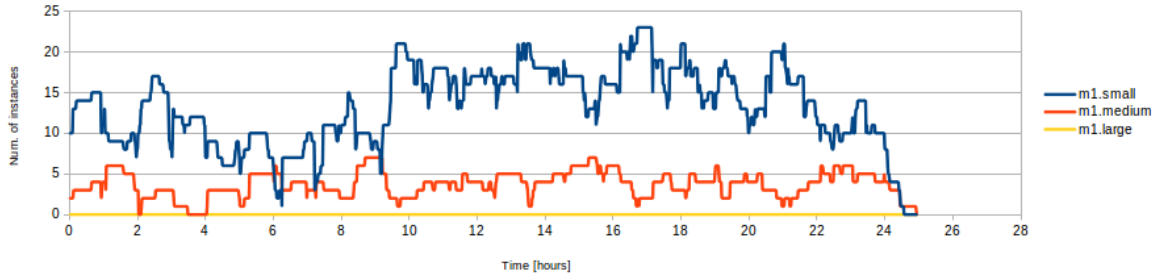
Figure 6.3: Test case scenario 1. Incoming load curve, scaling curve and instance type usage curve of region 1.



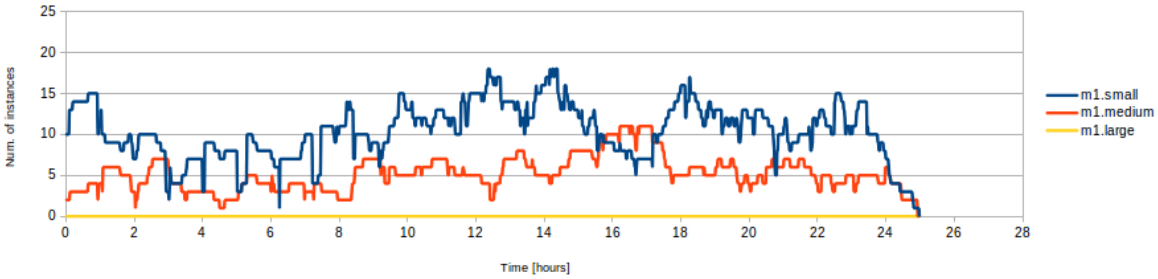
(a) Incoming load curve and scaling curve - Optimal



(b) Incoming load curve and scaling curve - Mini-optimal

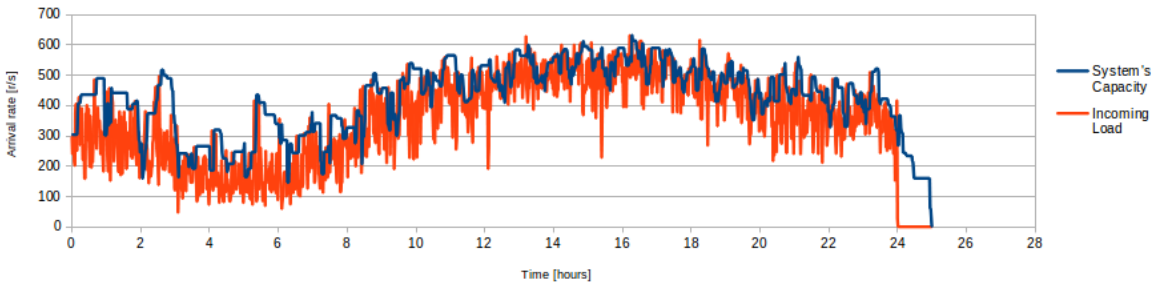


(c) Instance type usage curve - Optimal

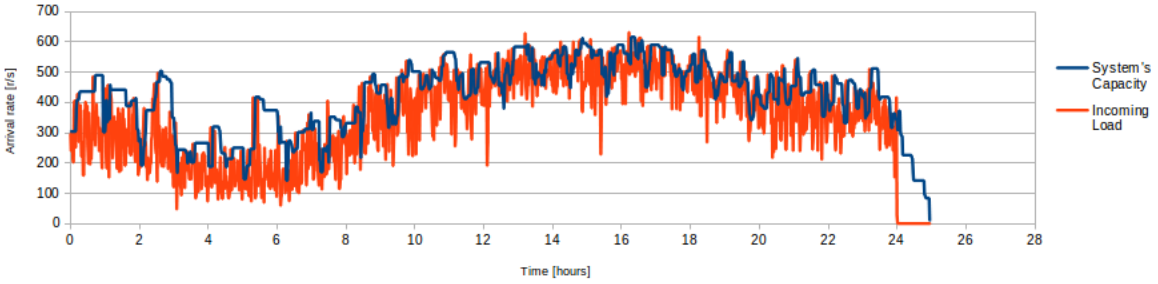


(d) Instance type usage curve - Mini-optimal

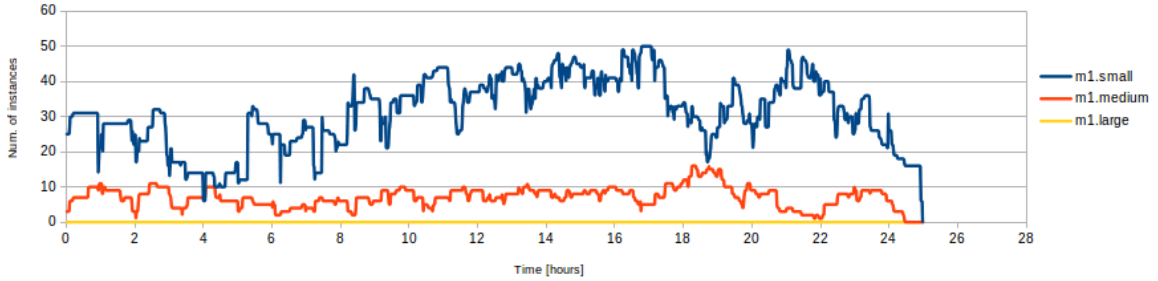
Figure 6.4: Test case scenario 1. Incoming load curve, scaling curve and instance type usage curve of region 2.



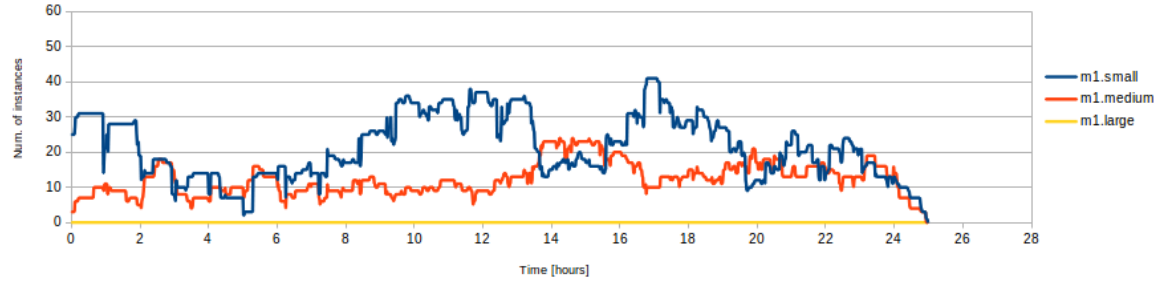
(a) Incoming load curve and scaling curve - Optimal



(b) Incoming load curve and scaling curve - Mini-optimal

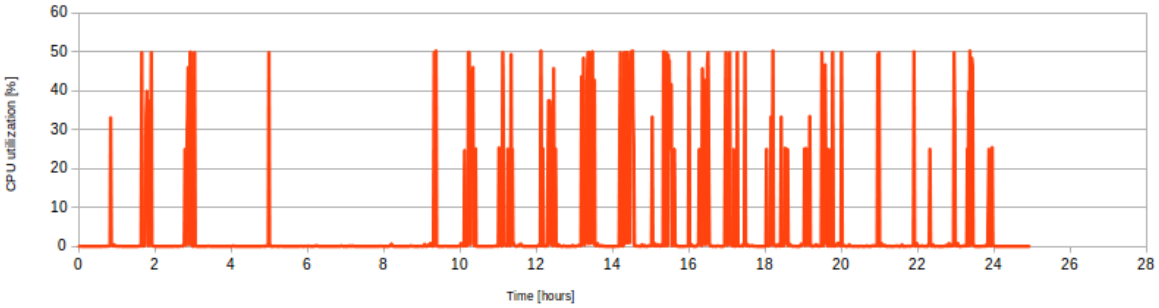


(c) Instance type usage curve - Optimal

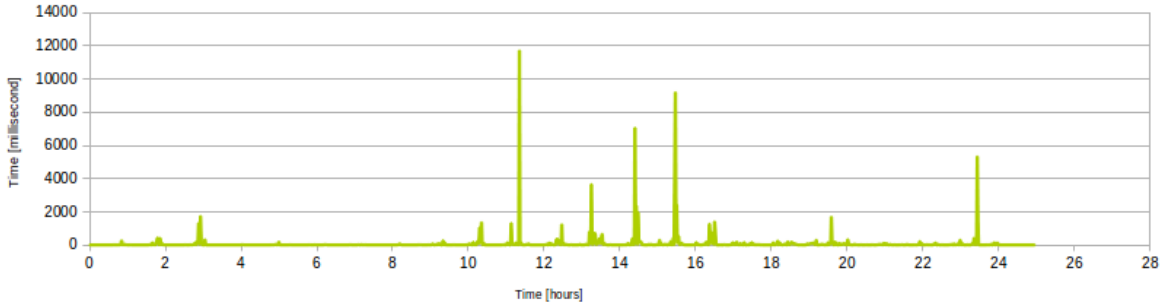


(d) Instance type usage curve - Mini-optimal

Figure 6.5: Test case scenario 1. Incoming load curve, scaling curve and instance type usage curve of region 3.



(a) CPU usage



(b) Time consumption

Figure 6.6: Test case scenario 1. CPU usage and time consumption of Optimal policy.

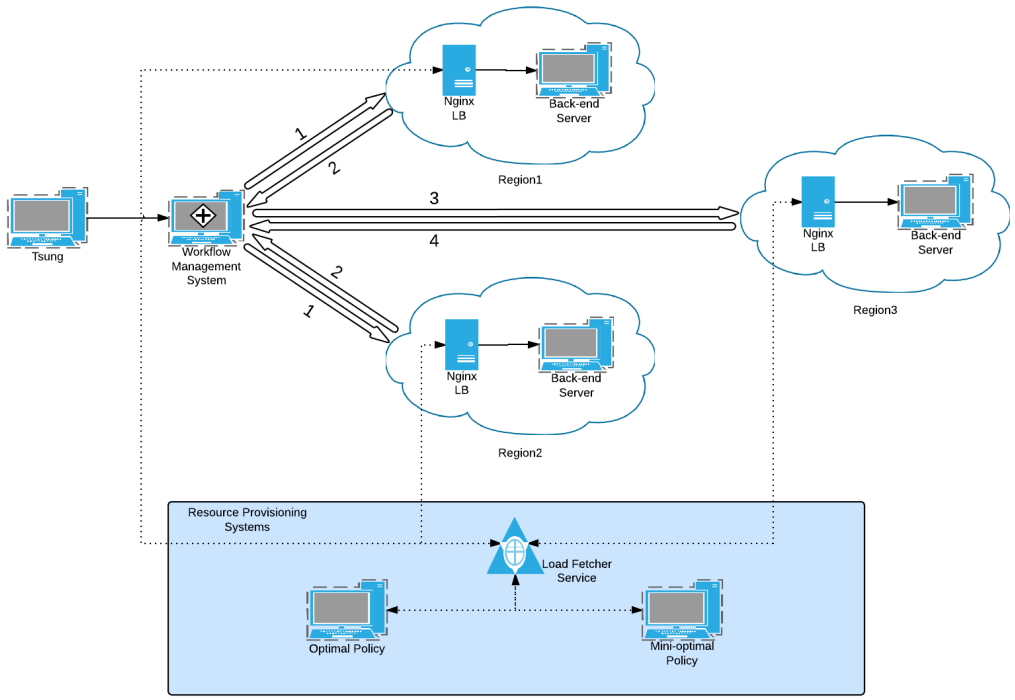
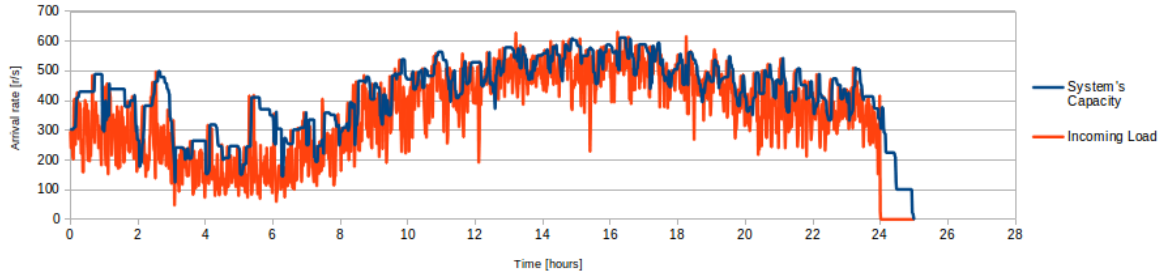
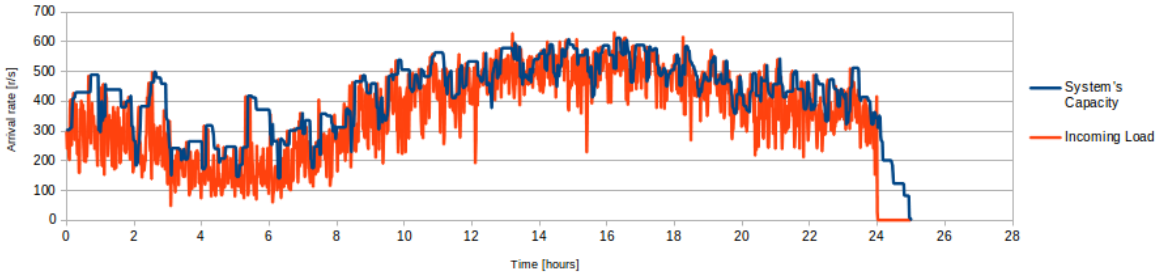


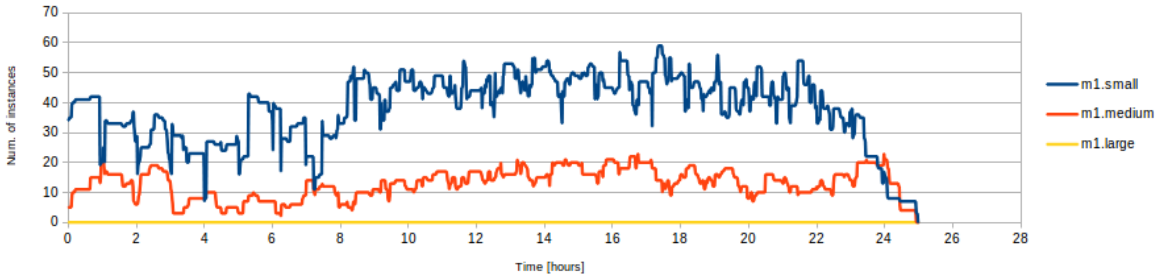
Figure 6.7: Test Case Scenario 2, with workflow management system, consisting of AND gate.



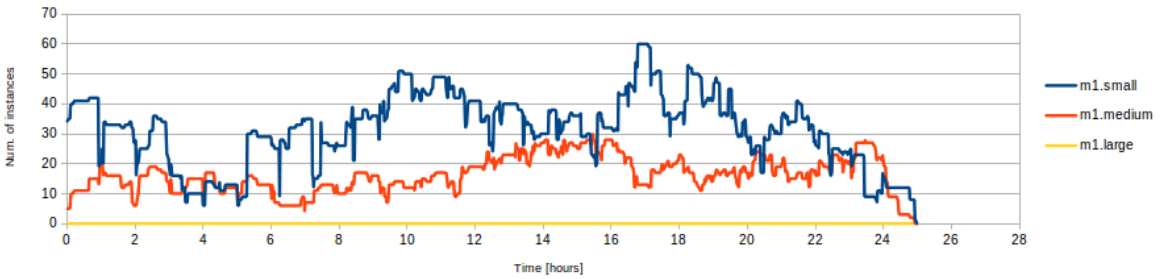
(a) Incoming load curve and scaling curve - Optimal



(b) Incoming load curve and scaling curve - Mini-optimal

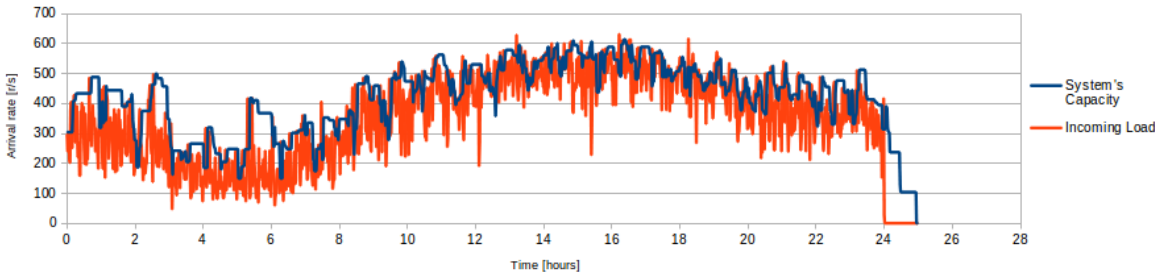


(c) Instance type usage curve - Optimal

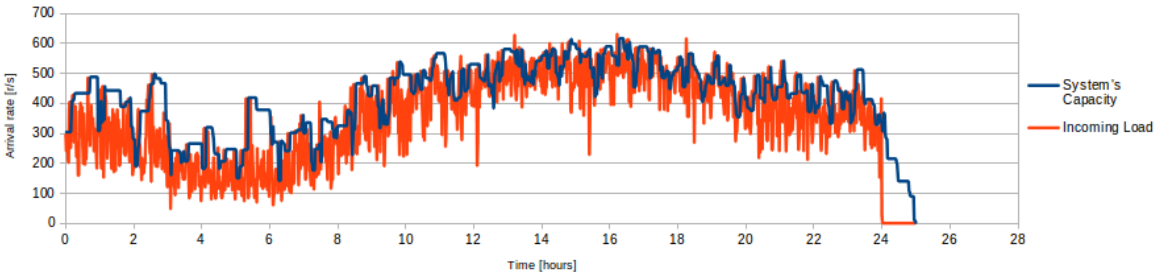


(d) Instance type usage curve - Mini-optimal

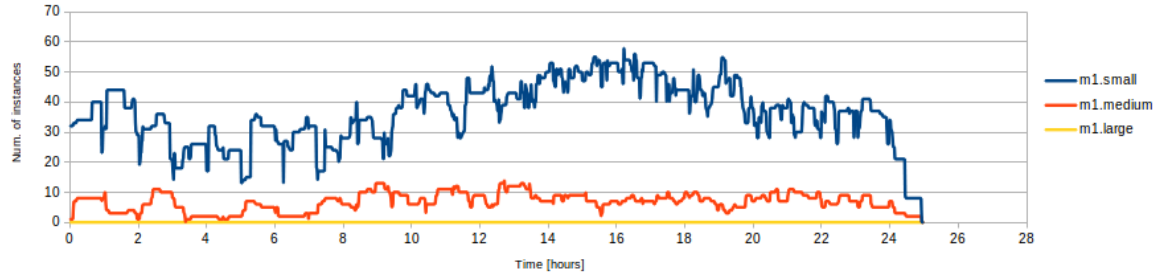
Figure 6.8: Test case scenario 2. Incoming load curve, scaling curve and instance type usage curve of region 1.



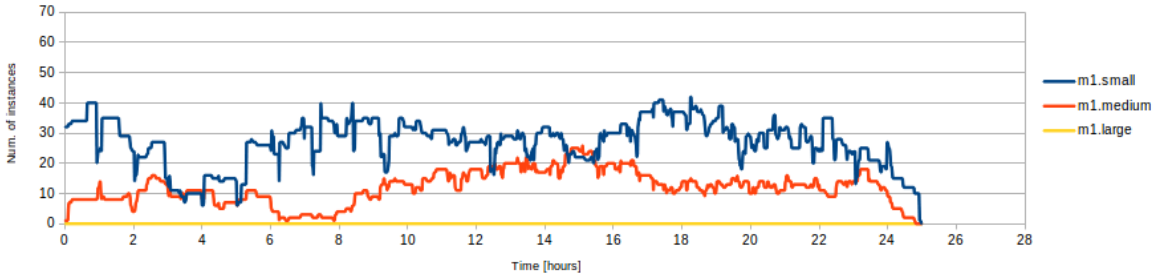
(a) Incoming load curve and scaling curve - Optimal



(b) Incoming load curve and scaling curve - Mini-optimal

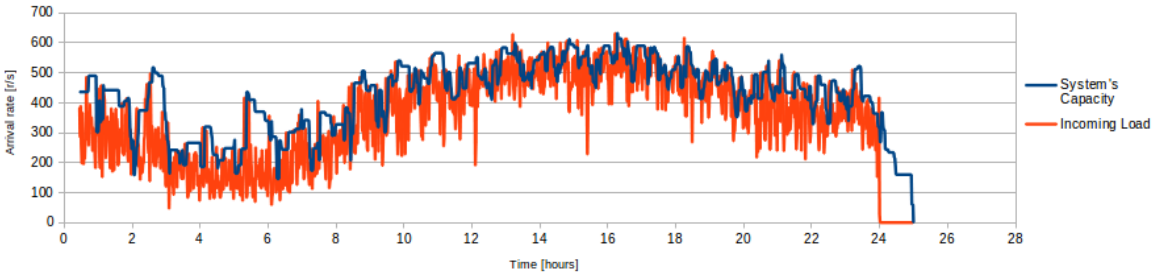


(c) Instance type usage curve - Optimal

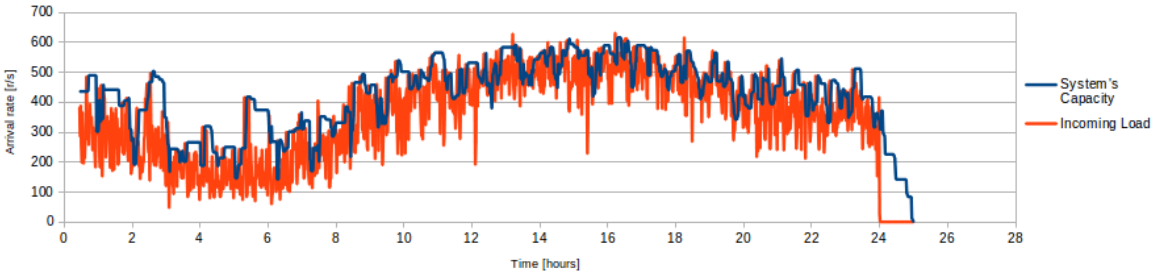


(d) Instance type usage curve - Mini-optimal

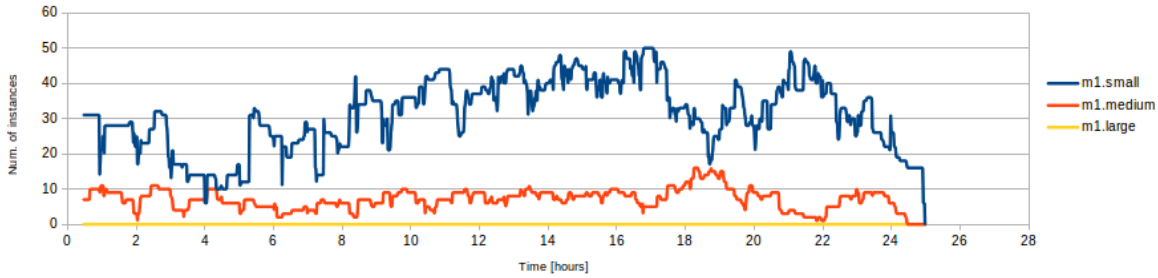
Figure 6.9: Test case scenario 2. Incoming load curve, scaling curve and instance type usage curve of region 2.



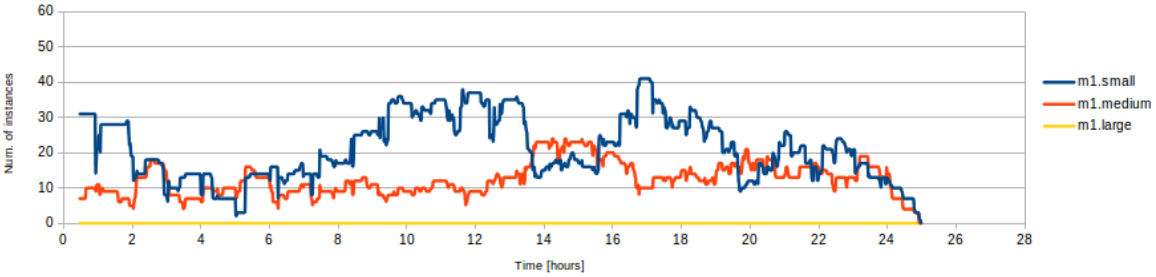
(a) Incoming load curve and scaling curve - Optimal



(b) Incoming load curve and scaling curve - Mini-optimal

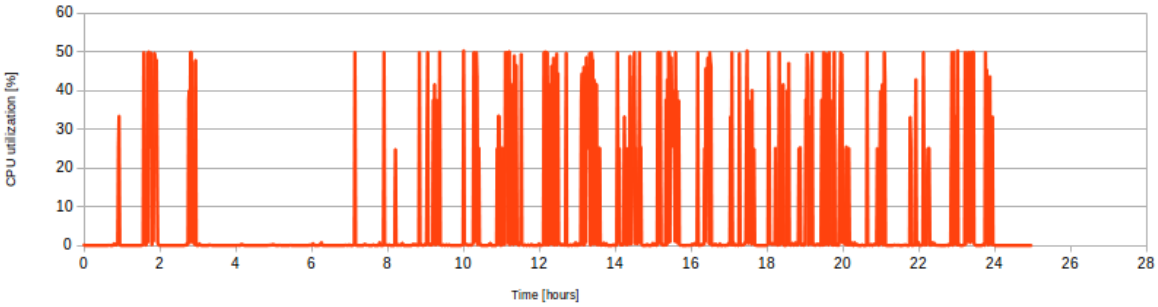


(c) Instance type usage curve - Optimal

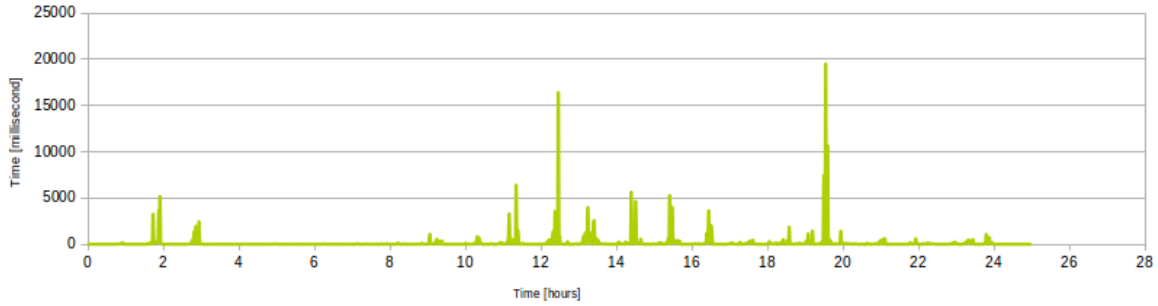


(d) Instance type usage curve - Mini-optimal

Figure 6.10: Test case scenario 2. Incoming load curve, scaling curve and instance type usage curve of region 3.



(a) CPU usage



(b) Time consumption

Figure 6.11: Test case scenario 2. CPU usage and time consumption of Optimal policy.

Chapter 7

Conclusion

In this paper, we introduced a novel resource provisioning policy that can find the most optimal setup of instances that fulfills incoming workload and minimizes the resource cost through taking full advantage of various available instance types. The presented LP model finds the optimal setup of each task in a workflow at each run. This model allows each task of the workflow to be hosted in a different cloud with different policies and instance types, suiting the task the most. All major factors involved in resource amount estimation such as processing power, periodic cost and configuration cost of each instance type and capacity of clouds are considered in our model. Additionally, the model takes lifetime of each running instance into account while trying to find the optimal setup, contributing to value determination of each instance at any time and discovering the optimal number of instances from each instance type in each region that must be added to or removed from the current setup. Using new concept of time bags and calculating two new costs bound to each running instance, namely killing and retaining cost, this method searches among all cost-effective configuration transformations, resulted from switching between various instance types having different processing power/price rates.

Two performance benchmarks were conducted on the model using a real load trace and through two main workflow management components, AND and XOR. In both experiments our generic LP model could find the most cost-optimal setup for each task of the workflow at any point of time within a decent amount of time. We showed that using time bags and by replacement of different-type instances we can still reduce the cost of running the system to some extent.

Chapter 8

Future Work

Even though the presented LP model considers most major parameters related to the scaling of a system in the cloud, there are still some parameters such as network bandwidth that can be added to the model. In data-centric applications, network bandwidth of the system plays the main role in scaling decision rather than processing power of the servers. So in contrast to the service-based application, in which we could mostly benefit from changing in number of instances, in data-centric applications we can mainly optimize running of the system by changing the network bandwidth.

As explained in chapter 5, our model is available to public through a restful web service hosted in heroku servers. However, currently we are working on a complete framework based on the model that can be used as the resource provisioning system of real-life applications, further increasing the usability of this novel optimal model.

Bibliography

[1] G. Juve, E. Deelman, Scientific workflows in the cloud, in: Grids, Clouds and Virtualization, Springer, 2011, pp. 71–91. 4

[2] A survey on decent witing time for loading a page. URL www.gomez.com 6

[3] Amazon EC2 Instances. URL http://aws.amazon.com/ec2/instance-types/ 9

[4] A. Ali-Eldin, J. Tordsson, E. Elmroth, M. Kihl, Workload classification for efficient auto-scaling of cloud resources. 13

[5] Hadoop. URL http://hadoop.apache.org 13

[6] J. Dean, S. Ghemawat, Mapreduce: simplified data processing on large clusters, Communications of the ACM 51 (1) (2008) 107–113. 13

[7] J. Wang, D. Crawl, I. Altintas, Kepler+ hadoop: a general architecture facilitating data-intensive applications in scientific workflow systems, in: Proceedings of the 4th Workshop on Workflows in Support of Large-Scale Science, ACM, 2009, p. 12. 13

[8] T. Lorido-Botrán, J. Miguel-Alonso, J. A. Lozano, Comparison of Auto-scaling Techniques for Cloud Environments, in: G. B. y Alberto A. Del Barrio (Ed.), Actas de las XXIV Jornadas de Paralelismo, Servicio de Publicaciones. Universidad Complutense de Madrid, 2013. 14

[9] T. Lorido-Botrán, J. Miguel-Alonso, J. A. Lozano, Auto-scaling techniques for elastic applications in cloud environments, Department of Computer Architecture and Technology, University of Basque Country, Tech. Rep. EHU-KAT-IK-09-12. 14

[10] Amazon Auto Scaling. URL http://aws.amazon.com/autoscaling/ 14

[11] Scalr. URL https://scalr-wiki.atlassian.net/wiki/display/docs/Home 14

[12] Rightscale. URL http://support.rightscale.com/ 14

[13] X. Dutreilh, N. Rivierre, A. Moreau, J. Malenfant, I. Truck, From data center resource allocation to control theory and back, in: Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on, IEEE, 2010, pp. 410–417. 15

[14] R. Han, L. Guo, M. M. Ghanem, Y. Guo, Lightweight resource scaling for cloud applications, in: Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on, IEEE, 2012, pp. 644–651. 15

[15] M. Z. Hasan, E. Magana, A. Clemm, L. Tucker, S. L. D. Gudreddi, Integrated and autonomic cloud resource scaling, in: Network Operations and Management Symposium (NOMS), 2012 IEEE, IEEE, 2012, pp. 1327–1334. 15

[16] Set up Autoscaling using Voting Tags. URL http://support.rightscale.com/12-Guides/Dashboard_Users_Guide/Manage/Arrays/Actions/Set_up_Autoscaling_using_Voting_Tags 15

[17] J. Kupferman, J. Silverman, P. Jara, J. Browne, Scaling into the cloud, CS270-Advanced Operating Systems. 15

[18] H. Ghanbari, B. Simmons, M. Litoiu, G. Iszlai, Exploring alternative approaches to implement an elasticity policy, in: Cloud Computing (CLOUD), 2011 IEEE International Conference on, IEEE, 2011, pp. 716–723. 15

[19] B. Simmons, H. Ghanbari, M. Litoiu, G. Iszlai, Managing a saas application in the cloud using paas policy sets and a strategy-tree, in: Proceedings of the 7th International Conference on Network and Services Management, International Federation for Information Processing, 2011, pp. 343–347. 15

[20] Reinforcement Learning. URL http://en.wikipedia.org/wiki/Reinforcement_Learning 15

[21] X. Dutreilh, S. Kirgizov, O. Melekhova, J. Malenfant, N. Rivierre, I. Truck, Using reinforcement learning for autonomic resource allocation in clouds: towards a fully automated workflow, in: ICAS 2011, The Seventh International Conference on Autonomic and Autonomous Systems, 2011, pp. 67–74. 16

[22] Queueing theory. URL http://en.wikipedia.org/wiki/Queueing_theory 16

[23] B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer, A. Tantawi, An analytical model for multi-tier internet services and its applications, in: ACM SIGMETRICS Performance Evaluation Review, ACM, 2005, pp. 291–302. 16

[24] Control theory. URL http://en.wikipedia.org/wiki/Control_theory 16

[25] H. C. Lim, S. Babu, J. S. Chase, S. S. Parekh, Automated control in cloud computing: challenges and opportunities, in: Proceedings of the 1st workshop on Automated control for datacenters and clouds, ACM, 2009, pp. 13–18. 17

[26] A. Ali-Eldin, J. Tordsson, E. Elmroth, An adaptive hybrid elasticity controller for cloud infrastructures, in: Network Operations and Management Symposium (NOMS), 2012 IEEE, IEEE, 2012, pp. 204–212. 17

[27] J. Xu, M. Zhao, J. Fortes, R. Carpenter, M. Yousif, On the use of fuzzy modeling in virtualized data center management, in: Autonomic Computing, 2007. ICAC'07. Fourth International Conference on, IEEE, 2007, pp. 25–25. 17

[28] M. Vasar, S. N. Srirama, M. Dumas, Framework for monitoring and testing web application scalability on the cloud, in: Proceedings of the WICSA/ECSA 2012 Companion Volume, ACM, 2012, pp. 53–60. 17

[29] C. Paniagua, S. N. Srirama, H. Flores, Bakabs: managing load of cloud-based web applications from mobiles, in: Proceedings of the 13th International Conference on Information Integration and Web-based Applications and Services, ACM, 2011, pp. 485–490. 17

[30] Linear programming.
URL http://en.wikipedia.org/wiki/Linear_programming 19

[31] OptimJ.
URL http://www.ateji.com/optimj/index.html 20, 35

[32] GLPK optimizer.
URL http://www.gnu.org/software/glpk/ 20, 35

[33] lpsolve optimizer.
URL http://lpsolve.sourceforge.net/5.5/ 20

[34] Cplex optimizer.
URL http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/ 20

[35] Mosek optimizer.
URL http://www.mosek.com/ 20

[36] Server virtualization.
URL http://searchservervirtualization.techtarget.com/definition/server-virtualization 21

[37] zenloadbalancer.
URL http://www.zenloadbalancer.org/web/ 22

[38] Nginx.
URL http://nginx.com/ 22

[39] Igor Sysoev.
URL http://sysoev.ru/en/ 22

[40] Nginx.
URL http://nginx.com/products/ 22

[41] PHP.
URL http://en.wikipedia.org/wiki/PHP 23

[42] I. NetEase, Apache 2.4.1 vs Nginx: A comparison under real online applications (2012). 23

[43] Amazon EC2.
URL https://aws.amazon.com/ec2/ 24

[44] Amazon Virtual Private Cloud.
URL http://aws.amazon.com/vpc/ 25

[45] Tools for Amazon Web Services.
URL http://aws.amazon.com/tools/ 25

[46] Amazon EC2 Instance IP Addressing.
URL http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-instance-addressing.html 26

[47] Amazon Regions and Availability Zones.
URL http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html 26

[48] Tsung.
URL http://tsung.erlang-projects.org/ 27

[49] Sigar.
URL http://www.hyperic.com/products/sigar 27, 49

[50] Amazon EC2 Instances.
URL https://aws.amazon.com/ec2/instance-types/ 43

[51] ClarkNet.
URL http://en.wikipedia.org/wiki/ClarkNet 44

[52] ClarkNet-HTTP.
URL ftp://ita.ee.lbl.gov/html/contrib/ClarkNet-HTTP.html 44

Non-exclusive licence to reproduce thesis

I, Alireza Ostovar, (date of birth: 22.09.1987),

1. Herewith grant the University of Tartu a free permit (non-exclusive licence) to reproduce, for the purpose of preservation, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright

Optimal Resource Provisioning for Workflows in Cloud,
supervised by Satish Narayana Srirama

2. I am aware of the fact that the author retains these rights.

3. This is to certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, 26.05.2014