FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master´s Thesis in Informatics

Scaling Policies Derivation for Predictive Autoscaling of Cloud Applications

Yesika Marlen Ramirez Cardenas

FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Scaling Policies Derivation for Predictive Autoscaling of Cloud Applications

Bestimmung von Skalierungsrichtlinien für Predictive Autoscaling von Cloud-Anwendungen

Author:	Yesika Marlen Ramirez Cardenas
Supervisor:	Prof. Dr. Michael Gerndt
Advisor:	Vladimir Podolskiy
Submission Date:	05.11.2018

I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 05.11.2018 Yesika Marlen Ramirez Cardenas

Acknowledgments

First of all, I want to thank my thesis advisor Vladimir Podolskiy for introducing me to the topic and for his guidance and engagement throughout the process of researching and writing this thesis.

Furthermore, I would also like to express my gratitude to Prof. Dr. Michael Gerndt for his feedback to the project and to the members of the research team who helped me with their contributions.

Finally, I would like to thank my family and friends for their support and continuous encouragement throughout my years of study and the process of this master thesis. This accomplishment would not have been possible without them.

Abstract

Cloud Computing enables the provision of services based on a pay-as-you-go model. Allocation and release of resources happen dynamically according to customers' needs following a Service Level Agreement (SLA). However, as there are a wide diversity of cloud applications and different business goals, the selection of a provisioning plan that adapts to the business dynamic becomes challenging.

Cloud applications may have an under-provisioning or over-provisioning problem since the Cloud Service Provider face the trade-off between minimize resources costs and meet the stringent Quality of Service (QoS) requirements. Known Auto Scaling solutions adjust the number of available computing resources according to the current service demand but they are either restricted to a single type of resource or they do not consider the type of cloud application to be scaled and do not anticipate workloads that arise at runtime which may cause SLA violations.

This thesis aims to improve the Auto Scaling process by deriving a set of scaling policies through the analysis of load patterns, application's performance profiles, resource types, and pricing model. The outcome of this research includes the implementation of the Scaling Policy Derivation Tool (SPDT) which based on different scaling strategies selects a cost-efficient scaling policy able to meet the current and predicted service demand.

The proposed solution helps Cloud Services Providers to efficiently manage the deployment and scaling of applications by answering questions like Which is the right number of resources needed?, what type or combination of type resources fits better?, what parameters and conditions should be considered before scaling?, When to start the scaling of resources?, how the type of application influence the scaling decisions?.

Keywords: Cloud computing, Auto Scaling, Predictive Scaling, Deployment Strategies, Scaling policies.

Contents

Acknowledgments **iii**

Abstract **iv**

1 Introduction **1**
- 1.1 Motivation 1
- 1.2 Research problem and Objectives 2
- 1.3 Proposed solution 2
- 1.4 Outline 5

2 Background **6**
- 2.1 Cloud usage scenarios 6
- 2.2 Technologies and paradigms 6
  - 2.2.1 Microservices-based Architecture 7
  - 2.2.2 Virtualization techniques: Virtual Machines vs Containers 7
- 2.3 Autoscaling Process 8
- 2.4 Scaling Challenges 8

3 Autoscaling policy building blocks **11**
- 3.1 Scaling indicators 11
  - 3.1.1 Indicators from a Cloud Service Provider perspective 12
  - 3.1.2 Indicators from a user perspective 12
  - 3.1.3 Hybrid 12
- 3.2 Scaling timing 13
  - 3.2.1 Reactive scalers 13
  - 3.2.2 Proactive scalers 13
  - 3.2.3 Hybrid 14
- 3.3 Scaling methods 15
  - 3.3.1 Vertical Scaling 15
  - 3.3.2 Horizontal Scaling 15
  - 3.3.3 Hybrid Scaling 16
- 3.4 Adaptivity 16
  - 3.4.1 Non-Adaptive 16

3.4.2 Self-Adaptive 17
3.5 Virtualization level 17
3.5.1 Virtual Machines 17
3.5.2 Containers 18
3.5.3 Hybrid 18
3.6 Pricing Model 18
3.6.1 On Demand 19
3.6.2 Reserved 19
3.6.3 Spot instances 19
3.7 Resource estimation 20
3.7.1 Rule based 20
3.7.2 Application profiling 20
3.7.3 Analytical modeling 21

4 Scaling Policy derivation 22
4.1 Input 22
4.1.1 Predicted workload as scaling indicator 22
4.1.2 Application Profiles 22
4.1.3 Virtual Machine Profiles 23
4.2 Output 23
4.3 Resource estimation 24
4.3.1 Estimation of pods 24
4.3.2 Estimation of Virtual Machines 25
4.4 Scaling strategies 26
4.4.1 Strategy: Naive 27
4.4.2 Strategy: Best resource pair 28
4.4.3 Strategy: Only Delta Load 29
4.4.4 Strategy: Always resize 30
4.4.5 Strategy: Resize when beneficial 32
4.4.6 Summary 33

5 Implementation of SPDT 35
5.1 Information Flow 35
5.2 Architecture overview 36

6 Experimental evaluation 41
6.1 Experiment setup 41
6.2 Evaluation metrics 42
6.3 Results analysis 43

6.4 Scaling plans comparison 45

7 Conclusions **51**
7.1 Limitations 52
7.2 Summary of Contributions 52
7.3 Future work 52

List of Figures **54**

List of Tables **55**

Bibliography **56**

1 Introduction

1.1 Motivation

Cloud computing has reshaped IT processes and enhanced the productivity of enterprises by provisioning IT services based on a pay-as-you-go cost model [1]. The outsourcing of tasks such as infrastructure maintenance or software acquisition not only helps users to save time and expenses but also it allows them to fully focus on their main business.

The wide range of offered cloud computing services is classified using a model that organizes them into a layered view that walks the computing stack from bottom to top [2]. Thus, the 3 major known categories are Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS), and Software-as-a-Service (SaaS). In addition, the 3 deployment models for accessing cloud computing environments are known as Public Cloud, Private Cloud, and Hybrid Cloud.

Adoption of cloud in many areas of application enables users to find the services that best fit their needs. Nowadays, the cloud hosts a diverse set of applications that range from online collaborative services to complex scientific processes [3]. As the cloud usage scenarios are wide, cloud providers are responsible for ensuring that Quality of Service (QoS) requirements are met.

The general autoscaling process can be described with the MAPE Loop, which consists of four phases: Monitoring, Analysis, Planning and Execution [4]. In the monitoring phase, data about the system is collected, then the gathered metrics are analyzed and used to validate predefined conditions or to make estimations of future resource utilization. Once the current or future state is known, a suitable resource modification (Eg. remove a VM or add more memory to a VM) is planned. The final phase is responsible for executing the deployment plan and adapting the resources' configuration.

Considering that the planning phase plays a major role in the implementation of any autoscaler since it takes the decision of how to scale the resources assigned to the application, the planning phase is the main focus of this project. The motivation for this thesis is then to improve the autoscaling process by developing a tool that uses simple heuristics to rapidly derive scaling policies for short and long terms. Finally, the outcome of this project is intended to be integrated into the implementation of a Predictive Auto Scaling Engine (PAE) which as a whole performs the autoscaling process and therefore it has specialized components for the other phases of the process.

1.2 Research problem and Objectives

Provisioning of resources in cloud platforms is dynamic, the assignation of resources change according to the configuration specified by the users. This configuration usually includes upper and lower bounds to define the maximum and minimum amount of resources that can be assigned to an application. Unfortunately, this approach is limited to a single resource type and does not take into account future workloads which may impact the application performance in case of significant load changes.

Moreover, some attempts to scale resources based on resource utilization predictions have been carried out such as the cases presented in [5]. However they do not take into account the application to be scaled and as Jindal, et al. presented in [6], autoscaling performance varies according to the type of application. (eg. a single or multi-layered application). Additionally, the specific time to trigger scaling operations and the step size used to expand or shrink resources are important variables into the equation as well, thus as Netto, et al. suggested in [7] autoscaling strategies that are not properly configured may lead to unacceptable QoS and large resources waste.

This thesis aims to explore and derive scaling policies for resource provisioning in a cloud environment. Thus, the research problem can be summarized by the following question: How to select a cost-efficient policy to scale a cloud application under a dynamic workload?.

As a result, this thesis pursues the following objectives:

- Identify the parameters and variables used to characterize different autoscaling policies.
- Propose different strategies to derive scaling policies for cloud applications.
- Develop a software tool able to select and schedule a cost-efficient scaling policy. In addition, the tool should be easily integrated into the implementation of the Predictive Auto Scaling Engine.
- Evaluate through a use case the proposed scaling strategies.

1.3 Proposed solution

The main outcome of this thesis is the implementation of a tool to derive scaling policies for predictive autoscaling of cloud applications. In order to achieve this, it is necessary to define and compare different scaling strategies that specify how to scale the resources assigned to a specific application. A sample of such strategy to derive policies is as follows: First, to forecast the amount of requests R for the application. Second, to identify the maximum amount of request r that each VM type T can handle. Then, to divide the predicted amount of requests by the identified maximum number of requests that can be served for different VM types. $n_T = \frac{R}{r_T}$ As result, each scaling policy would consist of a configuration determined by the number n of VMs of type T that can handle this many requests. Finally, to select the most suitable policy the total configuration cost could be evaluated for each policy.

The strategy presented to derive those scaling policies is a naive method based on a horizontal scaling approach. However, as it is a complex problem, more variables should be taken into account and a deeper analysis is required. Therefore, the implementation of the proposed solution is divided into four parts as is presented below:

1. The first part is to analyze different scaling approaches and identify their underlying techniques, parameters or models commonly used to build a scaling policy. Some scaling approaches focus on deciding what modification in resources is necessary for instance, vertical scaling improves performance by adding more physical resources, that is to say, memory, storage, and CPU to an existing virtual machine. In contrast, horizontal scaling increases the total capacity by adding a new virtual machine to a cluster in such a way that all the together function as a single unit.

   Other approaches focus on the time to start scaling. For example, reactive scaling responds to the system's current status, contrary to proactive scaling that uses sophisticated techniques to predict the future demand and adapt the provisioning of resources with enough anticipation. The expected output of this step is the identification of the main building blocks to design a scaling policy as well as the parameters used to take the scaling decision.

2. The second part consists of designing different scaling strategies to derive policies. The scaling approaches should consider the characteristics that indicate when to scale and what resource modification is required. Thus, the underlying technology used to deploy the application adds more variables that increase the complexity. For example, in the scope of this project, virtual machines are used as well as virtualization at the operating system level by using Docker containers.

   The scaling strategies are designed using heuristics considering the possibilities resulting from the combinations shown in the following matrix. Then each option is further specified taking into account the policy building blocks identified previously.

	Horizontal	*Vertical*	*Hybrid*
VirtualMachine	–	–	–
Container	–	–	–

3. The third part is to develop the Scaling Policy Derivation Tool (SPDT) which implements the scaling strategies introduced previously. This tool should be easily integrated into the Predictive Autoscaling Engine (PAE) currently under development. Additional components from the PAE provide the required information to derive the scaling policies and schedule them.

   The figure 1.1 gives an overview of the core components of the PAE and highlight the interfaces between the SPDT and other PAE's components.

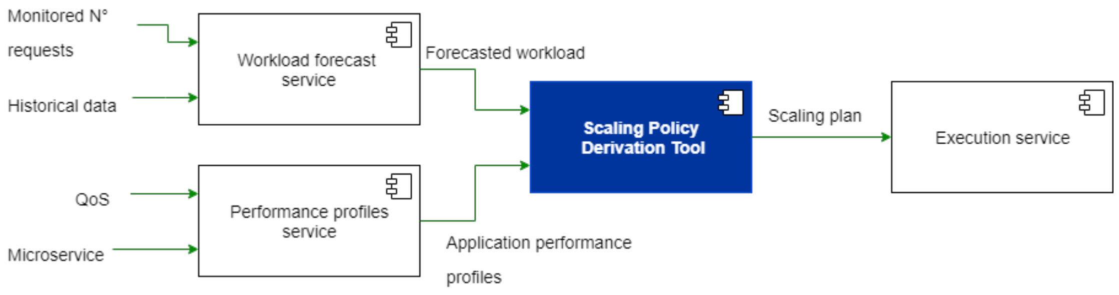

Monitored N° requests
Historical data
Workload forecast service
Forecasted workload
QoS
Microservice
Performance profiles service
Application performance profiles
Scaling Policy Derivation Tool
Scaling plan
Execution service

Figure 1.1: Overview of the Predictive Auto Scaling Engine

   - Workload Forecast Service: Component that forecasts the workload based on the historical information of the number of requests.
   - Performance Profiles Service: It builds performance profiles for the application that should be scaled as well as for the types of virtual machine where it is deployed.
   - Scaling Policy Derivation Tool: This component is developed in the scope of this thesis. It is responsible for deriving scaling policies and requesting its scheduling.
   - Execution Service: Component that schedules and executes the selected scaling policy for a time window.

4. The last part validates the proposed strategies to derive scaling policies and test the developed SPDT with 3 different types of applications and 3 different workload patterns. In order to compare the derived scaling policies, it is necessary

to define parameters that evaluate the quality of them. This helps to identify undesired scenarios, for example, a scaling policy that allocates or releases a virtual machine for every change in the workload without considering how long the new configuration would remain before a new load change appears, thus resulting in numerous deployments/undeployments. A sample of evaluation parameters includes the total cost of the policy, over-provisioning/under-provisioning, time between scaling decisions, etc.

1.4 Outline

The next chapter explains the background and key concepts related to the research problem. Then, the third chapter identifies parameters that characterize different auto scalers and presents a taxonomy with the building blocks to design an autoscaling policy. In the fourth chapter, five different scaling strategies are presented to derive a cost-efficient scaling policy. In chapter number five, the technical details of the implementation are described, followed by the evaluation of the scaling policies derived and a comparison of the scaling strategies used. Finally, the last chapter presents conclusions, a summary of the contributions of this thesis and future work.

2 Background

2.1 Cloud usage scenarios

The popularity of the cloud computing paradigm increases constantly given the multiple benefits provided, such as on demand provisioning, ease of use and the support for a wide range of application types. Authors in [3] presents a categorization for applications deployed in cloud environments to show the diversity of cloud-hosted applications, thus enabling the study of different cloud usage scenarios. From the proposed categorization we can highlight the following applications types and cloud usage scenarios: Messaging applications like e-mails and instant messages, applications for file storage and exchange like Dropbox, Google Drive or SkyDrive, Data mining applications that analyze datasets to discover relations between data such as Cloud9 Analytics or Google BigQuery, applications to store data in a structured manner such as the Database as a Service provided by Amazon SimpleDB. In addition, the education methodologies have changed by providing access to the information through Massive Open Online Courses like Coursera or edX. Another category groups the applications for User data processing such as Flickr. Moreover applications for entertainment such as audio/video streaming like Netflix and Spotify, and online gaming like World of Warcraft. Business processes improve the productivity by using applications for the Customer Relationship Management like Salesforce, Project and team management such as Huddle, and applications that enable the collaborative work like Wordpress. The industry of software development benefits from having cloud-hosted Development environments such as Kodingen as well as monitoring services and web hosting services. There are cloud applications that nowadays influence significantly user's behavior like social networking (e.g Facebook and Reddit) and Online commerce (e.g Amazon and eBay). Finally, there are available applications which offer services for scientific processes.

2.2 Technologies and paradigms

In order to carry out the wide range of cloud usage scenarios presented before, new methods to structure applications and for resource provisioning have been developed.

Each application type requires different strategies to be able to serve the increasing demand. The following subsections describe the trending Microservice-based Architecture for application development and deployment in the cloud, and the use of virtualization at the operating system level to optimize resource provisioning.

2.2.1 Microservices-based Architecture

In the traditional monolithic approach, all the functionalities are encapsulated into a single application. This approach in a first stage has advantages like simplicity to develop, test and deploy, however as the application grows it might become too complex to handle, therefore an approach that decomposes the monolith application into a set of small services and makes them communicate through a lightweight mechanism would overcome the challenge. This approach is called a Microservices-based Architecture and [8] proposes a dataflow-driven approach for effectively decompose a monolithic application into smaller services since it is still an issue that needs to be addressed in this architecture style.

Characteristics of a Microservices-based Architecture include the flexibility to scale the appropriate service instead of the entire full application and the capability of a seamless rolling out of new features. Moreover as shown in [9] microservices can help to reduce infrastructure costs in comparison to traditional monolithic architecture and even this approach enhances quality service measures such as service availability and service response through the performance optimization of resources provisioning [10].

2.2.2 Virtualization techniques: Virtual Machines vs Containers

Cloud computing services provide resources by means of virtualization. An approach is the use of virtual machines(VM) which somehow replicate a server, that is to say, that every VM includes a complete Operating System along with drivers, binaries or libraries [11]. Even though this allows to manage the resources more efficiently, when an application needs to scale, it might take more time than expected due to the abstraction layer overhead. A new strategy is the use of containers or so-called virtualization at an operating system level. Containers are lightweight and provide a faster start-up and low overhead compared to virtual machines. Containers technology increased its popularity with the launch of the Docker project. Docker is an open platform to develop, ship and run applications. It enables independence between applications and infrastructure, overcoming the challenges due to different dependencies per application required.

As the number of containers to be managed increase, orchestration solutions need to be included. One example is Kubernetes, a popular open source system created by Google to manage containerized applications. Its aim is to remove the complexity of deploying applications, how to decide where applications should run and how to ensure that they keep running. The basic unit of deployment in Kubernetes is called Pod, this is a wrap for one or more containers that share the same resources and local network. When a single pod cannot carry the load, Kubernetes can be configured to deploy new replicas of the pod to the cluster as necessary.

2.3 Autoscaling Process

The autoscaling process can be described using the MAPE Loop which consists of four phases. Monitoring (M), Analysis (A), Planning(P), and Execution(E).

- **Monitoring:** First, performance indicators are gathered to determine whether scaling actions need to be carried out. An autoscaling system requires a monitoring system that provides measures of metrics that describe not only the system's state but also the user's demand and the compliance with the expected QoS.
- **Analysis:** The analysis phase consists of processing the data gathered from the monitoring system and determine whether is necessary to perform scaling actions based on this information. In addition, the collected data can be used to predict future demands in order to arrange scaling actions with enough anticipation.
- **Planning:** Once the current or future workload is known, the autoscaler is in charge of planning how to scale the application. The planning involves precise times when the scaling action should start and the estimation of resources to provision or de-provision the resources assigned. In order to build the scaling plan is necessary to find a satisfactory trade-off between financial costs and QoS requirements.
- **Execution:** Finally, the execution phase is responsible for actually executing the scaling plan decided in the previous phase and provision or de-provision the resources. It is implemented through the Cloud Service Provider's API and its complexity consist of the ability to support diverse API's.

2.4 Scaling Challenges

Each phase has diverse challenges, which are listed here as the main questions that need to be answered to design an autoscaler. The decisions to scale a system rely on

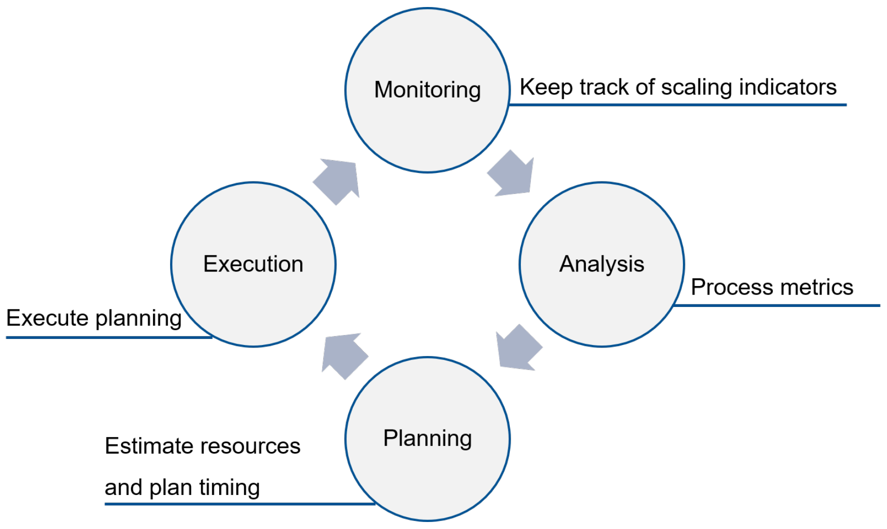


Figure 2.1: Autoscaling Process

having useful, updated and high quality performance metrics, therefore the decisions on the monitoring system are of vital importance to the success of the autoscaler.

- Which metrics should be monitored?. Depending on the type of application, some metrics would be more relevant than others. Monitoring every possible indicator is not feasible since monitoring also adds costs to the autoscaler system, and even some indicators could add noise to the relevant information, therefore a classification and prioritization of the metrics is required.
- How big are the monitoring intervals?. When the sampling granularity is too small, it might create a high overhead on computing resources and financial costs in order to obtain the metrics frequently. On the other hand, if the monitor interval is too big, the collected information might lack details useful that could be useful for the analysis.

Some of the open questions faced as part of the analysis include:

- When to perform the scaling actions? that is to say, whether the scaling actions should be carried out proactively or reactively. Moreover, in case that the proactive option is chosen, the next decision involves how to accurately predict the future workload?. The literature proposes several forecasting algorithms but the challenging task is to decide the most suitable for the system.
- How to adapt to changes?. Sometimes the workload presents unexpected changes which represent substantial modifications for the application. The autoscaling

system should be capable of adapting itself in a timely manner, process the new information, adjust models and set new configurations for the new situation.

- How to mitigate oscillations?. In order to avoid scaling actions that lead to a waste of resources or QoS violations, the analysis should include strategies to prevent the instability of the system.

Regarding planning, the main challenge is to estimate how many resources are just enough to handle the workload. The estimation needs to consider the deployment model, the amount and types of resources available, pricing model, etc. All these variables form a huge optimization space that needs to be solved efficiently in a short time.

3 Autoscaling policy building blocks

Autoscaling implementations differ in how the resources are assigned to an application. In order to derive a scaling policy, several characteristics that shape the nature of an autoscaler need to be considered, such as what metrics should be monitored?, when to to take the decision to scale?, what method to use to perform the scaling?, how the costs affect the scaling decisions?, how to handle changes on the workload?, how the customer influences the scaling decisions?, how transparent is for the user the decisions that are taken by the autoscaler?, etc. After an extensive literature review, this work lists a set of building blocks to derive a scaling policy and presents use cases as examples for each building block.

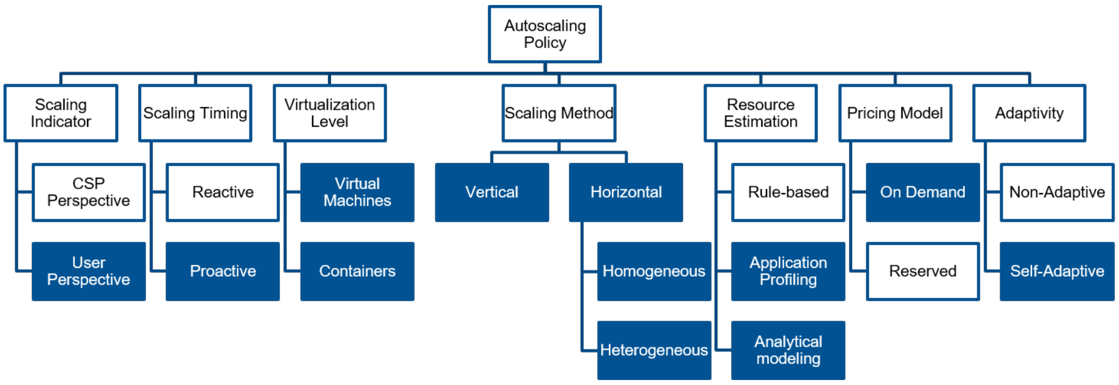


Figure 3.1: Autoscaling policy building blocks

3.1 Scaling indicators

The first step is to decide which metrics are relevant to take decisions. These indicators are collected at different levels and their analysis and importance depend on different perspectives. From a Cloud Service Provider (CSP) perspective the monitored metrics are mostly low-level and application agnostic, in contrast to the user's perspective where the indicators are mostly high-level and application dependent.

3.1.1 Indicators from a Cloud Service Provider perspective

This set of metrics consist of server information collected at low-level by the hypervisor. Currently, containers orchestration systems like Kubernetes or CSP like Amazon rely mainly on this infrastructure related metrics such as CPU utilization, which seems to be the most common metric in the design of autoscalers such as the cases presented in [12], [13] and [14]. Additionally, Fang et. al propose in [15] a provisioning schema which takes CPU utility, I/O rate, and network utility when the migration of services between nodes of a cluster is required. Furthermore, the case presented in [16] adopts a strategy that aims to minimize VMs' configuration cost considering the migration delay when a VM must be shut down and data is transferred to a new VM. Then, indicators like the size of the image, bandwidth and the booting time of a VM, along with factors such as Maximum Tolerable Delay are taken into account.

As pointed out by Botran in [4], the performance of a scaling policy depends on the quality of the available metrics, the sampling granularity, and the overhead cost to obtain them. Therefore, these metrics should be adapted accordingly to the use case and specific indicators could be used in the scaling decision. For example speed of disk access, memory usage, CPU-time per process, page faults, transmitted bytes, total thread count in an application server, number of transactions in a particular state in a database, average number of jobs in a queue, job's queuing time, etc.

3.1.2 Indicators from a user perspective

These indicators are observed at an application level and their analysis depends on the business logic. This implies a higher complexity level compared to the previous approach since it requires a deep understanding of the application. To accurately estimate these metrics, probably some off-line application profiling is required. An example of this type of metric is the workload defined in terms of the number of requests as Sedaghat et al. showed in [17]. Chieu et al. propose in [18] a dynamic scaling algorithm based on the number of active sessions and list as typical scaling indicators for web applications the number of concurrent users, number of requests per second, and the average response time per request. Other cases as the general autoscalers presented in [19] consider the response time, request arrival rates, and the average number of requests that can be served per VM per unit time to infer the number of resources.

3.1.3 Hybrid

It is not a trivial task to derive a scaling policy only considering metrics from one perspective since each approach has a drawback. On the one hand, the CSP perspective approach can not ensure that SLA is met since, from this perspective, application performance application is not visible. On the other hand, the user's perspective approach ignores the server information which is relevant to estimate the assignation of resources.

Therefore, approaches that combine both perspectives are presented in works like the cases presented by Fernandez in [20]. This is an example of the use of low and high-level indicators such as response time, request rate and % CPU utilization to build a profile of the virtual machines used to scale. In a similar manner, Taherizadeh et al. present in [21] a dynamic multi-level autoscaling method that uses infrastructure and application level monitoring data to scale applications in a container-based environment. In that scenario the monitored metrics given as input to the autoscaler include CPU usage of the cluster, memory usage of the cluster, application throughput in the current interval per container, application throughput in the last interval per container, application throughput in the second last interval per container and current number of running container instances in the cluster.

3.2 Scaling timing

When to start scaling is a critical decision since there is always a delay between the time from when a scale decision is taken until when it is effective. This delay may impact significantly the quality of service and costs. The approaches to decide when to scale are grouped into reactive autoscalers which depend on the current application's state and proactive autoscalers which define the timing with anticipation according to future needs.

3.2.1 Reactive scalers

The system reacts to changes in the current state. Those changes are detected using the last values from the set of monitored variables and consequently, the provisioning of resources is adjusted. However as the resource provisioning takes some time, it causes SLA violations being this the main drawback of a reactive autoscaler. Definition of thresholds is a widely used technique to trigger scaling actions, the user selects a metric and defines an upper threshold to increase the number of VMs and a lower threshold to reduce the number of VMs.

3.2.2 Proactive scalers

Proactive autoscalers take early scaling decisions. Two common proactive approaches are scaling based on scheduling or predictive scaling. Assignation of resources in the former is scheduled explicitly by the user. For example, for services that are only used in working hours, a suitable setting would be to schedule the booting of new VMs every morning at 7:00 am and schedule the release of them after 18:00 in order to guarantee a cost-effective use of the resources. In the later, the focus is to predict the workload or resources demand in future by using historical data and forecasting techniques.

Prediction techniques

The predictive autoscaling relies on the accuracy of the prediction algorithm. Extensive research has been done to predict application's workload, such as the work presented by Alipour et al. in [14], which propose a novel Microservices-based Architecture and forecast future CPU demand applying two machine learning models, Multinomial Logistic Regression and Linear Regression. After aggregate the prediction results from the two models, it produces a category from the three CPU usage target groups, namely High, Normal and Low.

Moreover, time series analysis is the most dominant approach in the cloud workload prediction area. For example, authors in [15] introduce a resource prediction and provisioning scheme which represents CPU utilization as a time series and employs a load prediction algorithm based on the ARIMA model. Likewise, the study case in [22] presents a predictive framework for virtual machines provisioning and introduces 3 time-series approaches including moving average (MA), autoregression (AR), and autoregression integrated moving average (ARIMA). In addition, authors compare the results predicting workload by using neural networks (NN) and the typical supervised learning method of support vector machine (SVM) which creates a classifier with a maximal margin based on the linear hyperplane.

3.2.3 Hybrid

The workload can be predicted when patterns can be found, however, is not feasible to forecast unexpected workload bursts. In this cases, the practical solution is a combination of proactive and reactive approaches as the work presented by the authors in [23] and [24]. This approach takes as input common rules based on thresholds and transforms the scaling strategy into a proactive approach to support changes in workload. The proposed framework has a reactive controller that is subscribed to the relevant monitored data required and makes scaling decisions based on the evaluation of rules. As complement, there is a predictive controller that uses 3 prediction models, one using a time series forecaster and two incrementally updateable Naive Bayes models. The predictive models learn on-line with each new data value, and eventually are able to take over the reactive controller.

3.3 Scaling methods

Methods to scale an application are known as horizontal scaling and vertical scaling. The first one refers to the strategy of adding or removing processing nodes to a cluster of virtual machines, in contrast, vertical scaling means adding or removing resources, like CPU, memory RAM, I/O, and network, from or to the existing virtual machines where the application is deployed. Each approach has its advantages and limitations, and its availability and implementation depends on the Cloud Service Provider, for example, the popular CSP Amazon Web Services only offers the out of the box option of horizontal scaling. These methods are also found in the literature as scale-out to indicate horizontal scaling and scale-up as a reference for vertical scaling [25].

3.3.1 Vertical Scaling

An advantage of vertical scaling is that it does not incur in extra overhead when it operates finegrained scaling at the resource level itself (CPUs, memory, I/O, etc). When handling virtual machines, a redistribution of physical resources is easily done, for example, if two VMs are hosted in the same physical server, the underutilized assigned resources of one VM can be quickly transferred to the other VM on demand [26]. In the same manner, when handling applications deployed in containers, the setting of CPU and memory limits can be increased to scale up the application without incurring any cost on either the cloud provider or the application owner.

3.3.2 Horizontal Scaling

Horizontal scaling is the most common method used to ensure elasticity in cloud computing. Despite the overhead of adding/removing machines, it is not limited by the physical resources of a single server like the case in vertical scaling. When considering a provisioning plan that uses horizontal scaling, it is important to decide the kind of cluster. On the one hand a configuration of resources with a homogeneous cluster is easy to implement choosing one VM type from predefined configurations offered by the CSP, on the other hand, a heterogeneous cluster could provide a more cost-efficient solution for some use cases. However, to find the optimal combination of VM types to build this kind of clusters implies a challenging task.

Homogeneous clusters

The use of homogeneous clusters of virtual machines is easy to manage. Usually, the Cloud Service Providers offer standard VM sizes or so-called VM types which have a predefined configuration such as the number of CPU cores and amount of memory.

When scale out is needed, the number of nodes increase preserving always the selected VM type.

Heterogeneous clusters

Making an optimal decision to build a VM cluster implies knowledge about the application, workload and the VM performance per type. The work presented by Lu et al. in [16] formulates the VM configuration problem as an optimization problem to determine the suitable VM types and the number of units of each type. Similarly, Sedaghat et al. in [17] consider how to efficiently provide the requested capacity considering the capacity of each VM Type and building a heterogeneous cluster.

3.3.3 Hybrid Scaling

According to the workload and the requirements of the application, use always one of the scaling methods described before would be enough. However, an approach where both techniques are combined could be feasible to ensure that the application is scaled in a manner that both, resource usage and reconfiguration costs are optimized. Authors in [27] state that using vertical scaling has a limited range due to the physical server limitations underneath but has lower reconfiguration costs, meanwhile, horizontal scaling can scale the application to a higher throughput despite a potentially higher cost. Therefore the authors suggest dividing the problem of resource provisioning into a first phase where the optimal size of the virtual machine is computed and a second phase to identify the minimum number of instances.

3.4 Adaptivity

The workload demanded is dynamic and sometimes unexpected and unpredictable. The capacity of an autoscaler to adapt its models or strategies from which the scaling decisions are made, is a desirable characteristic. The default option is however a non-adaptive autoscaler since it reduces the complexity.

3.4.1 Non-Adaptive

Using a non-adaptive approach, the autoscaler has a predefined model to take decisions. When no adaptivity is implemented, on the one hand, once a decision is made, it is not modified until the scaling action has already taken place. On the other hand, the model does not change its parameters or scaling indicators unless the user explicitly updates them. An example is illustrated with the rule-based approach employed by Amazon

Auto Scaling Service. Using this service, a threshold is defined by the user to trigger a scaling action. However, the threshold itself is static and it is limited to trigger a specific scaling action when a certain condition is met. Once the threshold is set, it does not change, unless the user updates it. In this case, the user must be aware of different scenarios to define different thresholds or to update them in case it is required.

3.4.2 Self-Adaptive

An autoscaler that implements a self-adaptive mechanism is capable of autonomously tunning its settings and update its decisions according to new incoming information. An example of a self-adaptable autoscaler is described in [13] by Liao et al. In this case of study, the authors propose the following strategy to dynamically adjust the thresholds for the CPU utilization and handle the creation of virtual machines according to the workload demand. Once the thresholds are set by the user and there is a significant increase on the workload, the upper threshold decreases linearly by 0.5% when the proportion of virtual machines increases by 1% with the purpose of minimizing the workload and avoiding unnecessary creation of machines. In contrast, the lower threshold increases linearly by 0.5% when the number of released virtual machines increases by 1% and when unreleased virtual machines increases by 1%, the lower threshold is reduced by 0.5% proportionally. This is done with the objective of reducing it to a value when the machines are no longer idle.

3.5 Virtualization level

Virtualization techniques consist of abstracting hardware resources as an approach to enhance resource management [28]. One approach is the use of hypervisor-based virtualization which abstracts hardware by adding a layer on top of the host machine and provide to the user access to the so-called virtual machines. A virtual machine runs a full abstraction of the operation system granting a high degree of isolation, although it increases overhead and reduces performance. As an alternative, Container-based or operating system level virtualization provides a lightweight consolidation, isolation and quick provisioning of resources.

3.5.1 Virtual Machines

The Cloud Service Provider tries to optimize resource allocation and power consumption, but at the same time provide access to the user to the requested resources such as virtual machines. The work presented in [29] integrates application autoscaling and dynamic virtual machine allocation in order to satisfy the needs of both cloud provider and the cloud user. Another example is presented by Chieu et al. in [18] which introduces a dynamic scaling algorithm based on the number of login users into a web application for automated provisioning of virtual machines.

3.5.2 Containers

As the workload changes dynamically, the use of virtualization based on containers provides a mechanism that improves application performance and resources utilization by enabling independence between applications and infrastructure. The container-based virtualization level increased its popularity with the launch of the project Docker. However, even though Docker keeps the container technology easy to port and replicate across different environments, most of the autoscaling solutions presented in the literature focuses mainly on the provisioning of virtual machines. Examples of dedicated scaling solutions that include characteristics of container-based virtualization include approaches like the work presented in [12], which is an autoscaling framework for containerized elastic applications. The proposed strategy scales containers vertically by provisioning more CPU when is required to quickly meet the resource demand, moreover scaler de-provision resources simply by reducing the numbers of containers.

Another case of study is presented by Klinaku et al. in [30] and this solution consist on scaling container units depending on their processing capability and manage an extra buffer of spare containers to handle unexpected workload bursts.

3.5.3 Hybrid

Most of studies address scalability of virtual machines and other few works focus on the scalability of containers. Even though a common practice is to deploy containers on top of virtual machines rather than in bare metal servers, there are few study cases that present the estimation of resources using a combination of both to scale an application. An example is presented by Al-Dhuraibi et al. in [31] which performs a combined effort to coordinate vertical scaling of both virtual machines and Docker containers.

3.6 Pricing Model

Multiple pricing models are available in the market. It highly depends on the available options offered by the Cloud Server Provider. As an example, this section presents some pricing options provided by Amazon Web Services, specifically for the EC2 Instances.

3.6.1 On Demand

The cost of computing capacity is charged per hour or second depending on the VM instance type. It does not require a long-term contract and the capacity is increased or decreased as the demand changes. The billing is based on pay-as-you-go according to the type of service and billing unit. Recently, AWS introduced billing-per-second which takes off the bill of the unused minutes and seconds. Even though the price rates have as a reference the hourly rate, the instances are charged only for the actual time in which they were used. Most of the study cases that focus on cost optimization assume that allocated resources can be used and paid for any period of time. For specific cases, the scaling actions would need to be aligned with the payment periods in order to avoid the unnecessary release of resources.

The billing unit has a significant impact on the cost efficiency for elasticity. A smaller billing unit like seconds offers more flexibility to decide on scaling actions, for example using billing per second, virtual machines can be terminated every time that the workload decreases, reducing the overall cost, however using billing per hour there is no advantage in terminating machines before the end of the period that was already charged.

3.6.2 Reserved

AWS EC2 reserved instances enable to reserve capacity. They are often used for applications that have a workload that does not have unexpected spikes. According to AWS, with this payment option the user can save up to 75% comparing to the on-demand option, however, this pricing model requires a commitment of minimum 1 year. Estimation of resources for long periods of time requires the expertise of the system administrator and the result would mostly lead to over-provisioning.

3.6.3 Spot instances

AWS EC2 Spot instances offer spare compute capacity that can be commercialized on markets through an auction-like mechanism an provide significant discounts in compared to on-demand instances. The work presented by Qu et al. in [32] focuses specifically on the use of heterogeneous spot instances and a pay per hour pricing model to scale web applications.

3.7 Resource estimation

How to estimate the number of resources and what scaling actions should be performed is a core issue in the design of any autoscaler. In the case of over-provisioning, extra costs are generated and if the assignation of resources is insufficient it may lead to SLA violations or to numerous scaling actions. The literature review presents several estimation models, from simple approaches to more sophisticated methods. The following subsections, present and explain some estimation models used in different use cases.

3.7.1 Rule based

The scaling process is triggered according to a predefined set of rules. For example, if the CPU utilization of all virtual machines is above 80% then increase the number of VMs and decrease it when the CPU utilization goes below 30%. Even though technically the specification of such rules are easy to implement, it is difficult to define the right boundaries in order to meet the tailored requirements for the application. Al-Dhuraibi et al. present in [33] an example of using rules to estimate the resources to scale containers vertically. The elastic controller adjusts memory and vCPU cores according to workloads. The upper and lower limits for thresholds are set based on experimentation by trying several values and selecting the ones that lead to less response time. Additionally, the ratios to increase or decrease CPU and memory are fixed. For instance, when the memory utilization is greater than the upper threshold it adds 256 Mb to the container and when is needed it decreases the memory by 128 Mb. A smaller scaling step is used for deprovisioning to avoid abruptly interrupt the functionality of the application.

3.7.2 Application profiling

Profiling an application is the process of measuring the saturation point of resources. For example, the space left in memory, the CPU utilization, or frequency and duration of a process. The profiles are used for understanding the application behavior and the goal is to use them for resource optimization. The simplest way to acquire the information is to perform an off-line analysis using statistical methods and simulations although the main disadvantage is that the profiling has to be performed every time that the application is updated.

3.7.3 Analytical modeling

This is the process of constructing mathematical models based on theory to describe the state or behavior of a system. This building block groups the approaches that describe the problem of scaling an application using formal methods. Following are some examples of such approaches.

Control theory

This approach creates a model of the application that includes a controller that provides feedback to automatically adjust the required resources. As described in [34] the feedback can be used to either satisfy a constraint or guarantee an invariant on the outputs of the application that needs to be scaled. The built-in mechanism of Kubernetes for autoscaling is an example of the use of control theory to estimate resources. In Kubernetes, a controller is a control loop that monitors the state of the cluster and makes changes attempting to move the current state towards the desired state, where a state is defined as the system configuration.

Queuing theory

It models the problem as a queue of requests in order to make a decision about the scaling action. In general, the queue can be represented as A/R/S, where A is the distribution of time interval between arrivals to the queue, R is the distribution of time required to process a request, and S is the number of virtual machines. An example of this model to estimate resources is presented by Han et al. in [35]. In this case, a greedy approach is used to add or remove one machine until the estimated capacity is just enough to serve the current load.

Reinforcement learning

This is a machine learning technique that enables an agent to learn in an interactive environment. The technique applied to the scaling problem attempts to learn the most suitable scaling action to be performed given a certain application state and incoming workload. The learning process follows a trial and error mechanism that uses as feedback old experiences. Once a scaling action is performed reinforcement learning uses rewards and punishments as signals to identify how good was the resource estimation. The main drawback is the required time to train a model from which the system can learn.

4 Scaling Policy derivation

This chapter presents 5 strategies to derive a scaling policy that capture the human understanding of the system, guarantee efficiency during runtime execution and comply to the system constraints such as budget and resources availability. The strategies were designed taking into account the building blocks described in the previous chapter. In particular, the blue blocks marked in 3.1 defined the base for the derivation of policies. In addition, the sections below describe the information used as input to take decisions and different mechanisms to estimate resources.

4.1 Input

4.1.1 Predicted workload as scaling indicator

From the user perspective, the workload can be described as the incoming number of requests. In case that the application is not able to serve the demand, the autoscaler should increase the capacity to avoid violations to the QoS. In the scope of this project, the predicted number of requests at time t_i is given as input and used as the main indicator to decide when to start a scaling action. Moreover, the policy is derived for the time window $[t\ ,\ t + \Delta]$.

4.1.2 Application Profiles

First of all, the policies derived are designed for microservices packed into containers orchestrated using Kubernetes. As a best practice, the resources assigned to pods in Kubernetes should be limited by specifying the CPU and the amount of memory RAM. Therefore, each application to scale has a performance profile depending on the configuration limits for the pods. An application profile received as input consist of the following attributes:

- Application's name
- Application type: Each type of application has a different capacity to handle the workload, therefore the scaling actions would change accordingly.
- CPU limits in pod configuration.

- Memory RAM limits in pod configuration.
- Number of pod replicas: Each application has a capacity **c** of the number of requests that can handle. A set of replicas provides the aggregated Maximum Service Capacity (MSC).
- Pods booting time: It is the expected time pull the docker image from the registry and boot the containers before the application is ready to receive requests.
- Maximum Service Capacity: Is the maximum number of requests that an application can handle without violating the QoS.

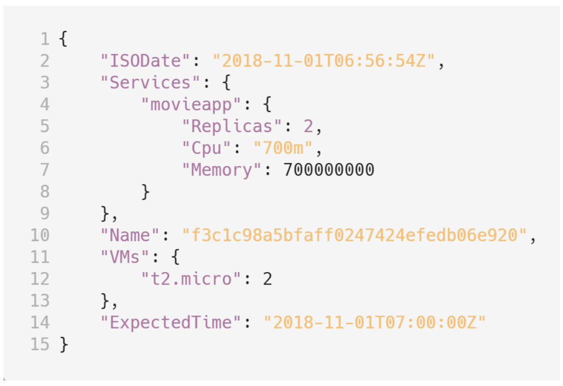
4.1.3 Virtual Machine Profiles

It describes the resources available to perform scaling actions, that is to say, the virtual machines and their configurations offered by the Cloud Service Provider. The following are the attributes considered in the profile of a virtual machine:

- Type of VM
- CPU assigned
- Memory RAM assigned
- Price per billing unit
- Billing unit
- Booting time
- Termination time

4.2 Output

A Policy derived by these strategies defines how and when to scale using a list of scaling actions. In addition, a state represents the configuration deployment for an application. In other words, the number of pods and their resource limits along with the number of virtual machines. Thus, each scaling action describes a transition between states. The figure 4.1 shows a JSON representation of a state, where ISODate is the timestamp of when the transition should start, and expected time is when the desired state is reached.



```json
{
    "ISODate": "2018-11-01T06:56:54Z",
    "Services": {
        "movieapp": {
            "Replicas": 2,
            "Cpu": "700m",
            "Memory": 700000000
        }
    },
    "Name": "f3c1c98a5bfaff0247424efedb06e920",
    "VMs": {
        "t2.micro": 2
    },
    "ExpectedTime": "2018-11-01T07:00:00Z"
}
```

Figure 4.1: Code Snippet: JSON representation of a State

4.3 Resource estimation

The scaling policies derived aim to scale microservices deployed using Kubernetes, therefore the way it manages resources is taken into account. Kubernetes considers two types of resources: CPU and memory RAM. It packs applications into pods and each pod has a limit on the assignation of resources. In addition, it manages a cluster of the virtual machines that provide the pool of resources to deploy pods on top of it.

In consequence, autoscaling requires coordination between two virtualization levels, using containers and using virtual machines. In the container level, we can use horizontal pod scaling and vertical pod scaling; and in the VMs level, we can scale up/down the number of nodes inside the cluster or replace VM types from one to another. The table 4.1 shows a reference to the symbols' notation used in the equations to estimate resources.

4.3.1 Estimation of pods

According to the application profiles described previously, there is a direct relation between the number of requests that a microservice can handle and the number of pods along with the resource limits assigned. Therefore, horizontal scaling and vertical scaling are the methods to estimate the new pods' configuration.

Horizontal pod scaling

Consist of increasing the number of pod replicas keeping the same resource limits configuration (CPU, RAM) per pod. An approximation to find the right number of

Symbol	Description
R	Number of requests
S	Set of Virtual Machines
T	VM type
K	Number of VM types
n_T	Number of VMs of type T
P_T	Price per time unit of one VM of type T
P_S	Total price for a set of VMs
P_R	Total price for the reconfiguration or transition between states
d	Time interval for which the VM set S is billed
L	Duple that represents the pod's resource limits (CPU,RAM)
L_{CPU}	CPU assigned in resource limits L
L_{RAM}	Memory RAM assigned in resource limits L
m	Number of pods
m_L	Number of pods using limits L
PC_T	Pod capacity or number of pods that a VM type T can host
MSC	Maximum Service Capacity

Table 4.1: Notation used in equations

pods m given a demand of requests R is shown by the equation (4.1) where MSC is the Maximum Service Capacity provided by a pod that has L as resource limits.

$$m_L = \left\lceil \frac{R}{MSC_L} \right\rceil \tag{4.1}$$

Vertical pod scaling

The objective is to update the number of resources assigned to each pod, in other words, to modify the CPU and memory RAM limits L. The information about the relation between MSC and L is given by the application profiles. Hence to decide about the new limits configuration when scaling vertically, a search on the profiles is required to find a pod's configuration able to serve the number of requests R with the minimum number of pods.

4.3.2 Estimation of Virtual Machines

In order to find an optimal VM set S of type T:

1. The first step is to calculate the maximum number of pods that a VM type T can host. This is referenced as pod capacity PC_T and is calculated as follows:

A VM has a resource capacity assigned by the CSP from which some resources are reserved for the installation of Kubernetes, thus $AllocableResources_T = NodeCapacity - Reserved$

Considering the resource limits L of a pod, the capacity depending on the CPU that can be assigned is $Assigned_{CPU} = \frac{Allocable_{CPU}}{L_{CPU}}$. In addition, the capacity according to the memory that can assigned is $Assigned_{RAM} = \frac{Allocable_{RAM}}{L_{RAM}}$

Then, the minimum value between the assignation of both resources is taken and this would be the pod capacity PC_T for a VM type T and pod's limits L

$$PC_T = Min(Assigned_{CPU}, Assigned_{RAM}) \tag{4.2}$$

2. Next, knowing the pod capacity, the number n of VMs type T is given by the equation below.

$$n_T = \left\lceil \frac{m_L}{PC_T} \right\rceil \tag{4.3}$$

cost calculation

The only resources taken into account for the billing are the VM instances leased from a CSP. According to the VM profiles received as input, each VM type T has a price P_T associated to a billing unit. For example, in the scope of this project billing per second was used and the calculation of the price for a VM set S is:

$$P_S = n_T * P_T \tag{4.4}$$

According to the CSP documentation, the billing for a VM instance starts not from the moment that the booting order is launched but from the moment when the VM is up and ready to use. In contrast, the termination of a VM takes time and the billing for that VM only stops when the VM is actually shut down.

4.4 Scaling strategies

All the strategies presented here consider first to scale at the level of containers and only in the case that it is necessary to scale the virtual machines. This is beneficial since only scaling pods does not generate extra costs. In contrast, when scaling VMs, the CSP bills every new VM added to the cluster.

4.4.1 Strategy: Naive

The naive strategy aims to increase or decrease the application's capacity by using horizontal scaling at both virtualization levels: Containers and Virtual Machines. It starts by taking the total number of requests R at time t and estimate the resources as follows:

1. Number of pod replicas: Using current pod resource limits L, query the application profiles to find the MSC provided by one replica. Then find the number m of pod replicas using equation (4.1).
2. VM set S: Using current VM type T calculate its pods capacity PC_T. Then find the number n of VMs T needed to host m_L pods as described in equation (4.3).

As a drawback, the cost-efficiency of this approach highly depends on the resource limit constraints assigned to the pod and VM type defined by the system administrator at the moment of the first application deployment. It means that the system administrator requires a high expertise and knowledge about the application in order to set an optimal configuration. The resource estimation used in this strategy is the simplest and it was included in the implementation mainly as a baseline for comparison given that is the most popular approach used by different CSPs.

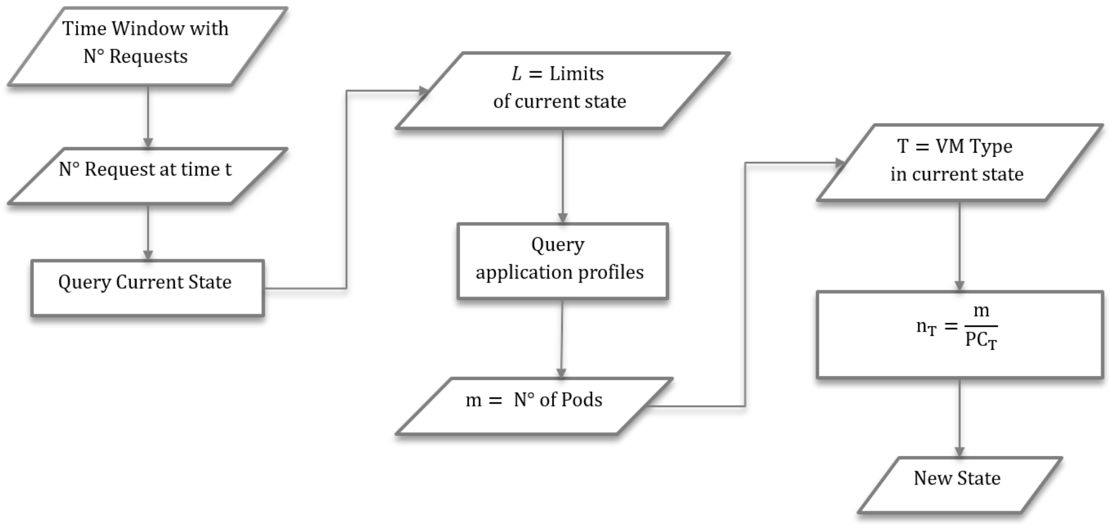

Time Window with N° Requests
N° Request at time t
Query Current State
L = Limits of current state
Query application profiles
m = N° of Pods
T = VM Type in current state
$n_T = \frac{m}{PC_T}$
New State

Figure 4.2: Strategy: Naive

4.4.2 Strategy: Best resource pair

The first deployment does not always have an optimal resource configuration, therefore this strategy aims to find it. Assuming that the initial state has a deployment configuration that consists of m pods with resource limits constraints L, hosted by a set S of n VMs. The goal is to find a new optimal resource combination for the incoming workload. A resource combination is defined as the pair (T, L) where T is the VM type used to build a VM set S and L refers to the resource limits constraint to configure the pods. A flow chart that summarizes the best-resource-pair strategy is given in figure 4.3.

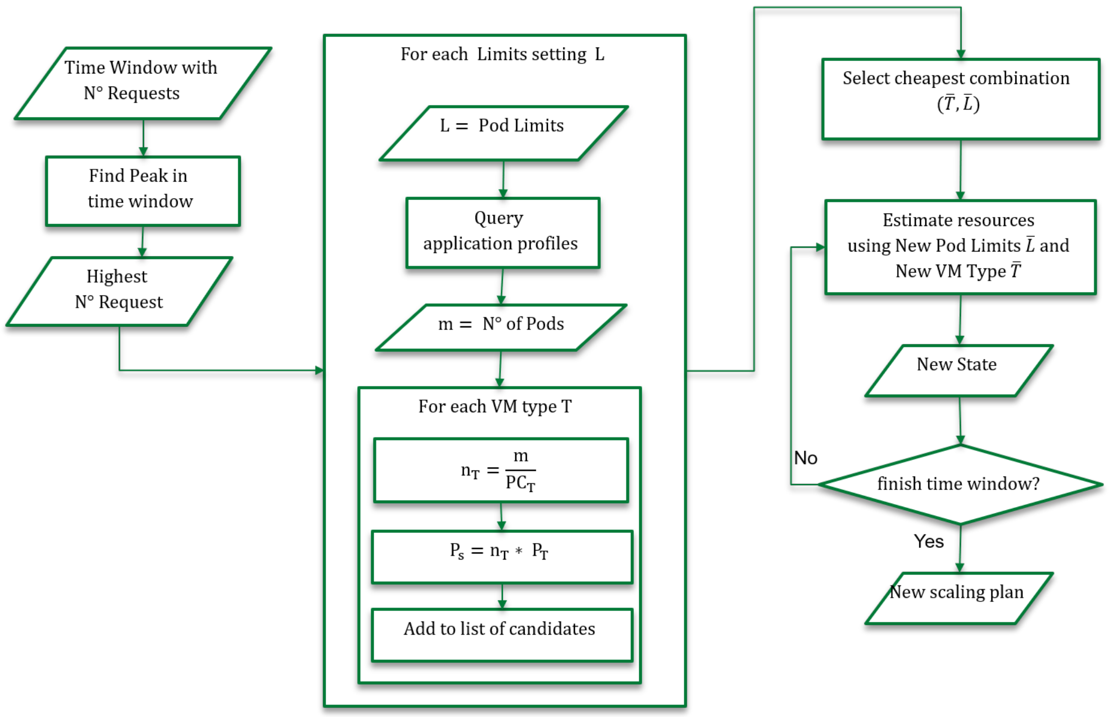

Time Window with N° Requests
Find Peak in time window
Highest N° Request
For each Limits setting L
L = Pod Limits
Query application profiles
m = N° of Pods
For each VM type T
$n_T = \frac{m}{PC_T}$
$P_s = n_T * P_T$
Add to list of candidates
Select cheapest combination $(\bar{T}, \bar{L})$
Estimate resources using New Pod Limits $\bar{L}$ and New VM Type $\bar{T}$
New State
finish time window?
No
Yes
New scaling plan

Figure 4.3: Strategy: Best-Resource-Pair

In order to find the best resource pair, the following heuristic is used:

1. Find the highest peak in the time series received as input, this would indicate the maximum number of requests that need to be served in the time window for which the policy is derived.
2. Extract from the application profiles a list of resource configuration limits L for pods and build combinations against the list of VM types T available.

3. Next, for each pair (T, L) find a set S that provides the capacity to host the pods. Then, the price of each VM set S is calculated using (4.4). and the cheapest one is selected. In addition, when there are two or more sets S with the same cost, the one that requires fewer pod replicas to handle the number of requests is selected.

Once the best resource pair ($\overline{T}$, $\overline{L}$) is found, a one time scaling action using vertical scaling is performed at the level of containers as well as at the level of virtual machines to change from the current configuration deployment with (n_T, m_L) to ($\overline{n_T}$, $\overline{m_L}$)

Then, from the next time t the estimation of resources continue using exclusively the best resource pair as the base and the method of horizontal scaling to increase/decrease capacity. The resource configuration shaped by this strategy over the time window consists of homogeneous clusters using VMs of type $\overline{T}$ and pods with resource limits $\overline{L}$.

4.4.3 Strategy: Only Delta Load

The previous strategies took as input, the *total Load* represented here as the number of requests at time t and found the appropriate number m of pods and the correspondent set S of VMs. The Only Delta load strategy on the contrary uses as input the *delta Load* at time t where *delta Load = total Load - current capacity*.

In case that delta load is > 0 and resources need to be increased, then it finds an optimal VM set $\overline{S}$ to satisfy specifically the *delta Load* demanded.

This approach does not perform any vertical scaling on the containers or any migration of virtual machines, the resources are scaled as described below:

1. Pods are scaled horizontally using equation (4.1).

2. Virtual machines are removed from or added to the previous allocation with the flexibility of selecting a different VM type T accordingly for different *delta Load*. For example, if the workload increases in small steps, a small VM is added.

Eventually, over time the result would be a set of several small VMs and different heterogeneous VM sets. A flow chart that summarizes the only delta load strategy is given in figure 4.4.

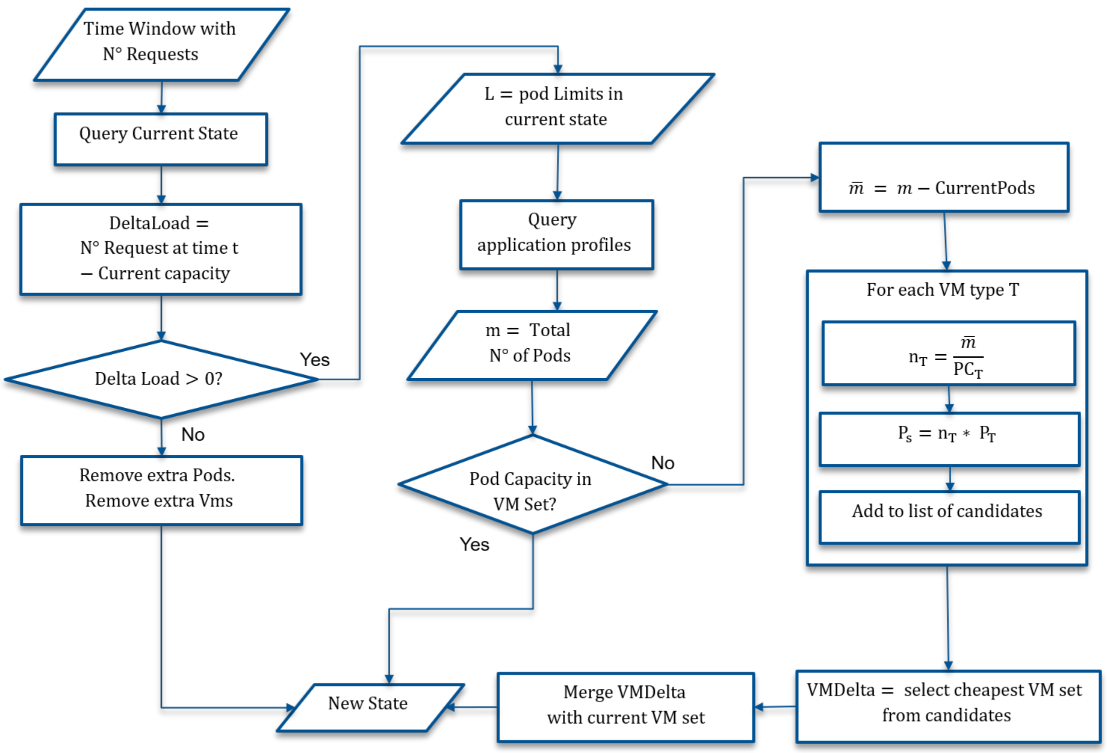

Time Window with N° Requests
Query Current State
DeltaLoad = N° Request at time t − Current capacity
Delta Load > 0?
Yes
No
Remove extra Pods. Remove extra Vms
L = pod Limits in current state
Query application profiles
m = Total N° of Pods
Pod Capacity in VM Set?
No
Yes
$\overline{m}$ = m − CurrentPods
For each VM type T
$n_T = \frac{\overline{m}}{PC_T}$
$P_s = n_T * P_T$
Add to list of candidates
VMDelta = select cheapest VM set from candidates
Merge VMDelta with current VM set
New State

Figure 4.4: Strategy: Only-Delta

The new VM set $\overline{S}$ added is optimal for the *delta Load* at time t although could be suboptimal with respect to the *total Load* .

4.4.4 Strategy: Always resize

In this strategy, reconfigurations using vertical scaling at the level of containers as well as at the level of virtual machines is explored. On the one hand, the different application profiles provide information about the Maximum Service Capacity (MSC) of an application using different resource limits constraints L. On the other hand, the CSP used in the scope of this implementation does not offer the option of actually modify the size of a VM, therefore vertical scaling at the level of VMs for this strategy consist in replacing the types of VM being used. Thus, a migration from one VM set S using the type T to another VM set $\overline{S}$ using a type $\overline{T}$ is performed when a VM with more resources (CPU, RAM) is required.

1. Estimation of pods: From the list of pod limits provided by the application profiles, find the pod limits L that would provide the highest MSC by using less

amount of resources. This is achieved by calculating the allocation/performance ratio defined in equation (4.5) for each L_i and selecting the L_i that has the smallest ratio.

$$Ratio_{allocation/Performance} = \frac{m_L * (L_{CPU} + L_{RAM})}{MSC_L} \tag{4.5}$$

2. Estimation of VM set: Once the number of pods m with limits configuration L have been selected, a new VM set $\overline{S}$ is found using equation (4.3).

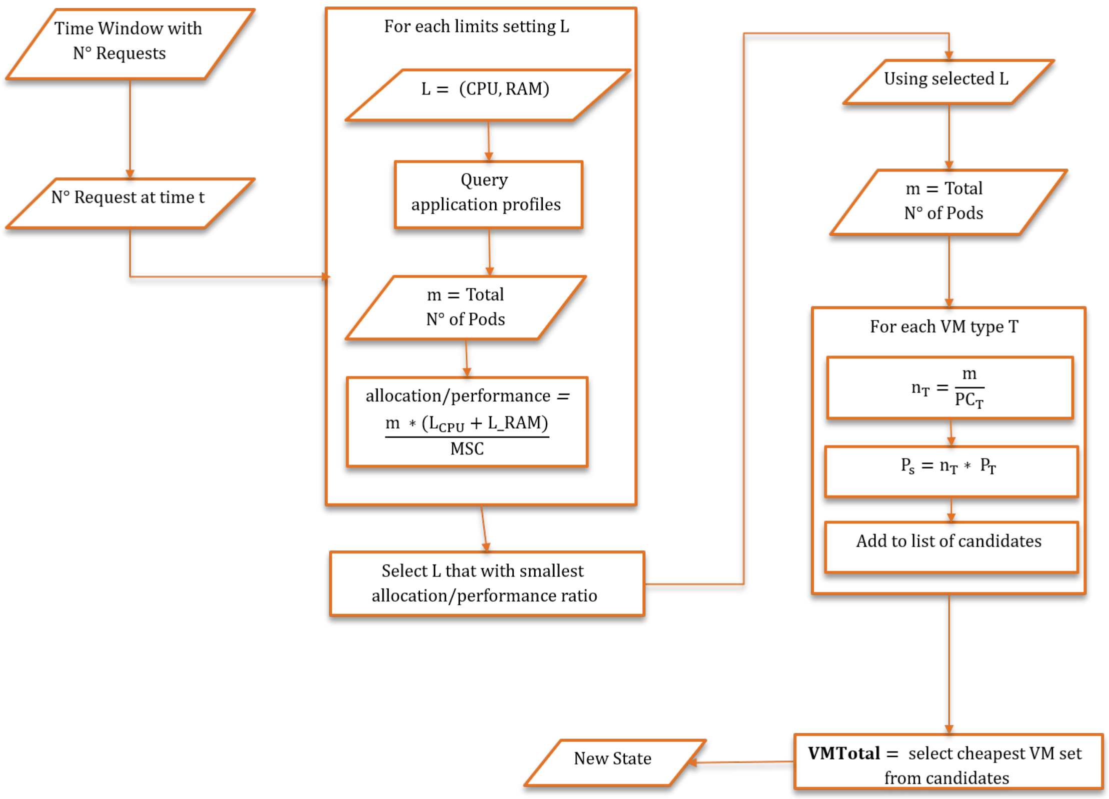

Time Window with N° Requests
N° Request at time t
For each limits setting L
L = (CPU, RAM)
Query application profiles
m = Total N° of Pods
allocation/performance = m * (L_CPU + L_RAM) / MSC
Select L that with smallest allocation/performance ratio
Using selected L
m = Total N° of Pods
For each VM type T
n_T = m / PC_T
P_s = n_T * P_T
Add to list of candidates
VMTotal = select cheapest VM set from candidates
New State

Figure 4.5: Strategy: Alway-Resize

This strategy not only adds extra capacity by adding new VMs but it may also replace VMs with a more cost-efficient type. However, to migrate or replace a VM type an overlapping between the initial VM set S and the desired VM set $\overline{S}$ is required. It means that during the transition of the two states both configurations would add an overhead on costs in order to meet the QoS.

4.4.5 Strategy: Resize when beneficial

The two previous strategies presented were agnostic of the impact of the scaling decision taken at time t and how this one influences the next scaling action. The aim of this approach is to include this awareness and select the option that is more beneficial. On the one hand to reconfigure at each time t the current state using vertical scaling to supply the demand of *total load* could be very expensive in the case that many VM migrations should be performed. On the other hand, to consider only the optimal configuration for *delta load* would be more expensive since each solution would still be suboptimal with respect to *total load*.

At some point in time, the set of VMs allocated would benefit from being replaced by a new VM set that provides the required capacity at a lower cost. Therefore based on the idea of repacking introduced by Sedaghat et. al in [17], this strategy is implemented. A flow chart that summarizes the *resize when beneficial* strategy is given in figure 4.6.

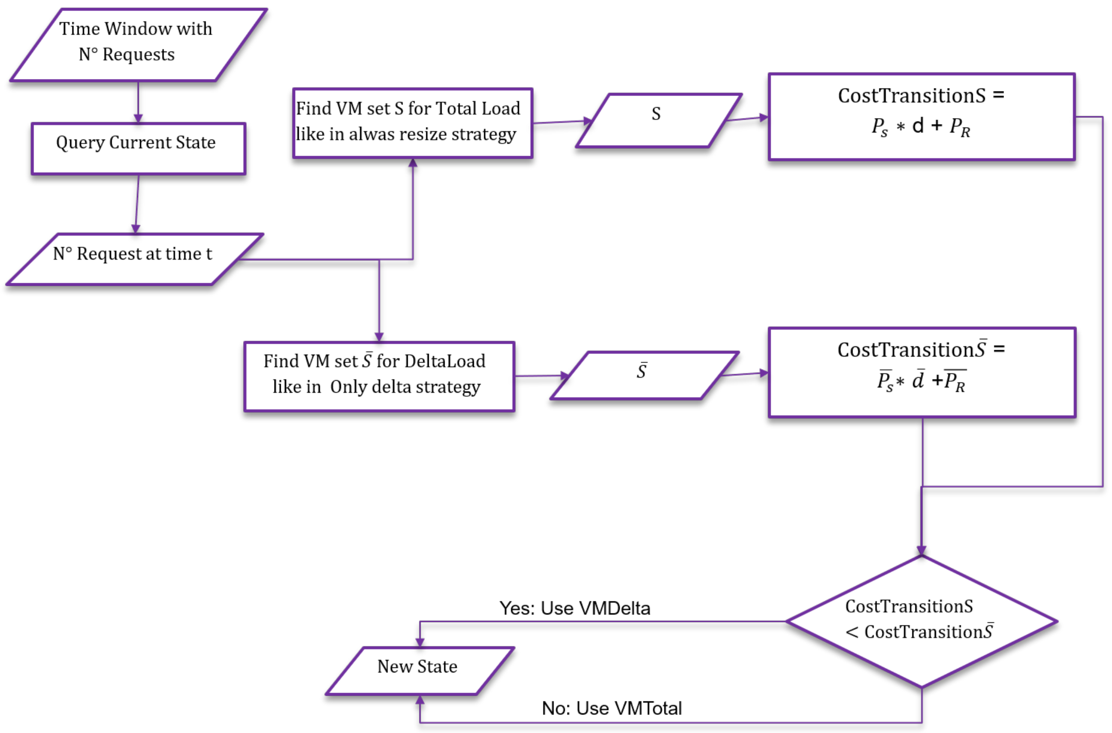

Time Window with N° Requests
Query Current State
N° Request at time t
Find VM set S for Total Load like in alwas resize strategy
S
CostTransitionS = $P_S * d + P_R$
Find VM set $\bar{S}$ for DeltaLoad like in Only delta strategy
$\bar{S}$
CostTransition$\bar{S}$ = $\bar{P_S} * \bar{d} + \bar{P_R}$
CostTransitionS < CostTransition$\bar{S}$
Yes: Use VMDelta
No: Use VMTotal
New State

Figure 4.6: Strategy: Resize when beneficial

At each time t the resource estimation is done by comparing the configuration deployment suggested as result of the *only delta load* strategy against the configuration deployment estimated using the *always-resize* strategy to handle the *total load*.

This strategy evaluates whether is worth it to use a vertical scaling approach, despite the potential overhead in costs when VM migration is required. In this case, there is an overlapping of VM set S configurations which increase the cost of the transition between states. To migrate from S to $\overline{S}$ it is necessary to keep the current VM set S running meanwhile the new VM set $\overline{S}$ boots and is ready to use, then the VMs in S are terminated.

The decision to migrate to a new VM type is taken based on the following comparison:

$$\overline{P_S} * \overline{d} + \overline{P_R} < P_S * d + P_R$$

where P_S is price of VM set S, d is the interval of time during which this configuration will be running before a new scaling action takes place, and P_R is the cost of reconfiguration. The calculation of P_R is done as shown by the equation (4.6).

$$P_R = \sum_{T=1}^{K} n_T * P_T * d \tag{4.6}$$

4.4.6 Summary

Regardless of the virtualization level, there are have two possible methods to scale resources, that is to say, either horizontally or vertically. With this in mind, the difference between the five presented scaling strategies lies in the combination of methods used to coordinately scale both containers and virtual machines. For instance, the *Naive* strategy uses horizontal scaling for both VMs and containers. In contrast *Always resize* strategy uses vertical scaling for both levels, either by adding more resources to the pod or changing the VM type for one with more resources. Moreover, examples of hybrid approaches are *Best resource pair* and *Resize when beneficial* strategies that scale resources horizontally or vertically depending on the convenience.

In addition, the types of available virtual machines to set up the cluster makes another important difference between the strategies. For example, the *Only delta* and *resize when beneficial* strategies are the only ones that consider the option of having a heterogeneous cluster of virtual machines, meanwhile the other approaches only allow clusters with machines of the same type during the same time interval.

The table 4.2 summarizes the inputs and main characteristics that differentiate the scaling strategies.

Strategy	**Input**	**scaling containers**	**scaling vms**	**cluster type**
Naive	Total load	Horizontal	Horizontal	Homogeneous
Best resource pair	Total load	One time Vertical then Horizontal	One time Vertical, then Horizontal	Homogeneous
Only delta load	Delta load	Horizontal	Horizontal	Heterogeneous
Always resize	Total load	Vertical	Vertical	Homogeneous
Resize when beneficial	Total load	Horizontal and Vertical	Horizontal and Vertical	Heterogeneous

Table 4.2: Summary scaling strategies

5 Implementation of SPDT

This chapter describes the implementation of SPDT, a tool to derive scaling policies for cloud applications. The logic behind the policy derivation is based on the scaling strategies presented previously and as mentioned in the introduction, this tool is integrated as a component in the implementation of a Predictive Autoscaling Engine (PAE). Therefore, in the following sections, the components of SPDT are described along with the interaction with other components of the PAE.

5.1 Information Flow

A typical information flow to derive a scaling policy is represented in the figure 5.1.

1. The process to derive the scaling policies starts by reading the configuration file Config.yml which helps to set up the initial settings such as endpoints to connect with the other components of the PAE, preferred scaling strategy, budget, etc.
2. Secondly, the communication with the other components of the PAE is done using the HTTP protocol, therefore an HTTP request is performed to retrieve the application profiles from the Performance Profiles Component of the PAE.
3. Next, the information about virtual machines regarding their hardware configuration and prices is obtained by reading the file vm_profiles.json provided by the user.
4. To continue, the information about virtual machines is complemented by retrieving the VMs' booting/shutdown times for all the VM types listed in vm_profiles.json. The information is fetched from the Performance Profiles Component and stored in the correspondent database.
5. Further, the number of requests for the time window specified in the config.yml file is fetched from the Workload Forecast Component of the PAE.
6. Moreover, once the forecasted time window was retrieved and stored, SPDT uses an HTTP request to subscribe with the Forecast Component to receive updates whenever the predictions for that time window change, thus the tool is capable of adjusting the scaling policy accordingly.

7. Since the Executor component of the PAE knows the current state of the configuration deployment, SPDT performs an HTTP request to get the current number of VMs and pods and take this information as a starting point to estimate future resources.

8. Then, using all the information retrieved and the heuristics used by the scaling strategies, the scaling policies are derived.

9. In addition, the scaling actions suggested by the selected scaling policy are scheduled by sending them to the Executor component.

10. Finally the information is stored and available for its visualization.

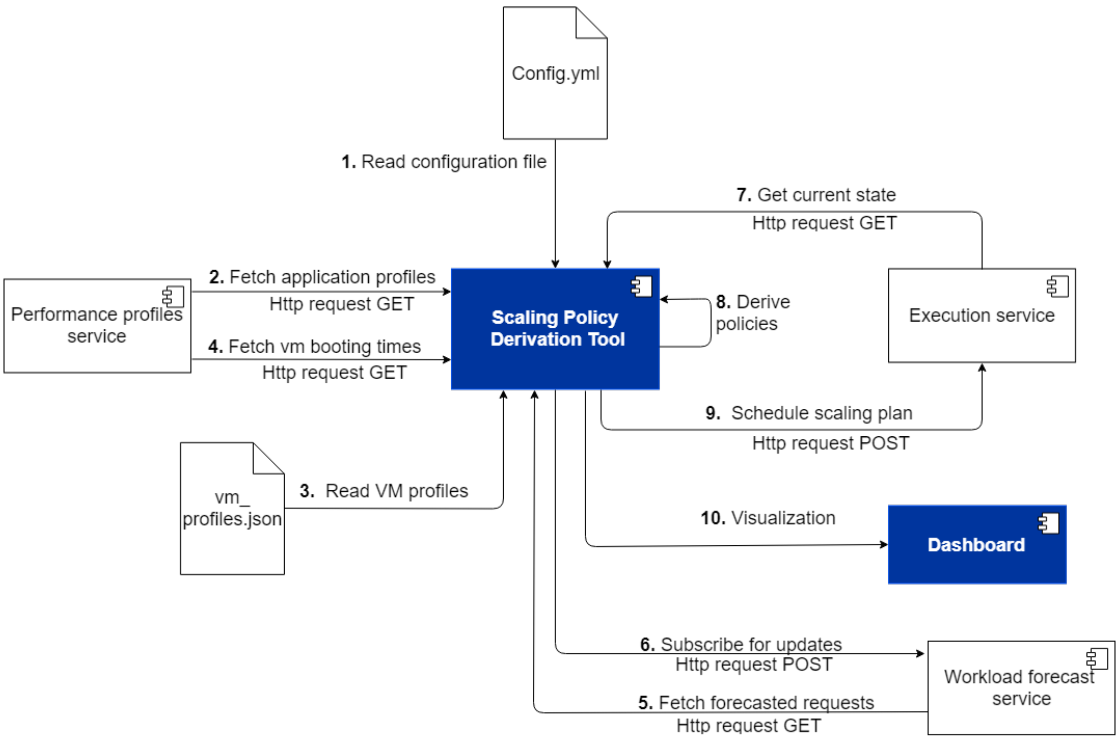


Figure 5.1: Information flow

5.2 Architecture overview

SPDT is implemented modularly and packed into docker containers in order to ease its deployment and portability. Figure 5.2 depicts an overview of the SPDT's components and a description of each one is presented below.

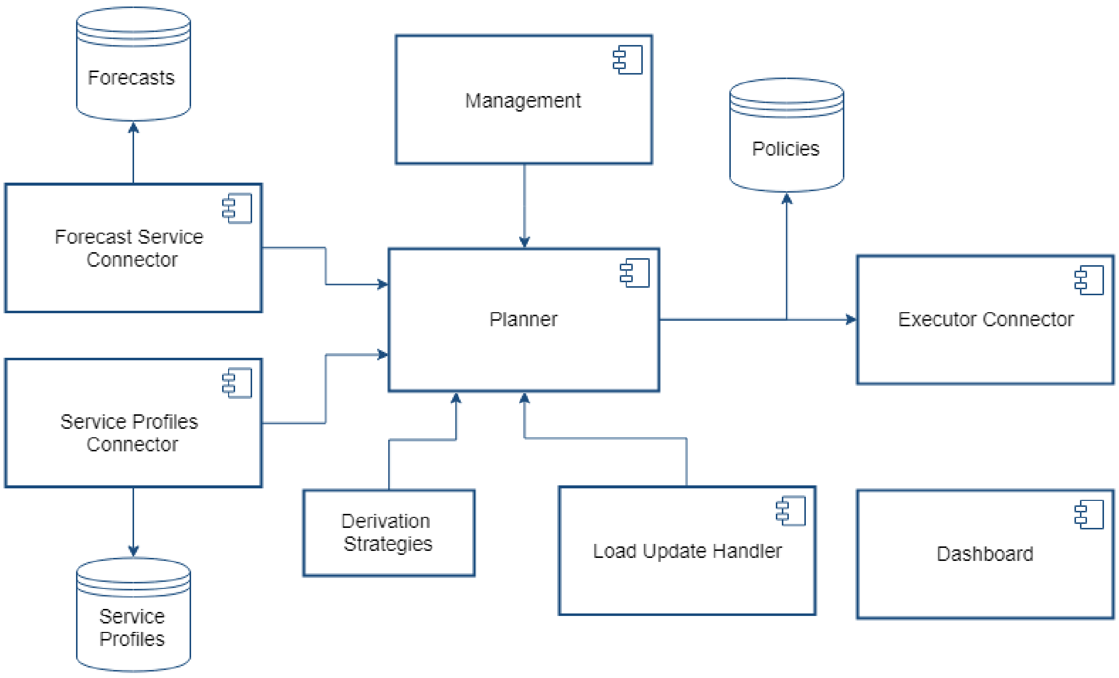


Figure 5.2: Overall architecture of SPDT

Forecast service connector

This component is responsible for pulling the predicted number of requests for a time window specified by the user. The pulling interval is defined as the frequency at which an HTTP request to the forecasting component of the PAE is performed. Its default value is a parameter that should be set by the user. Once the information is received, the time serie is processed and parsed into a list of intervals for which a scaling action should be carried out.

Forecasts database

The information is received from the workload forecast component in a JSON format that contains an id to identify the forecasted time window, an array of future timestamps with the expected number of requests. The information is stored temporarily in MongoDB and is deleted from the database when it becomes irrelevant for the time considered or when new values arrive. The irrelevance of an old forecasted time window is defined by the user by specifying the storage time interval as part of the initial configuration of the system. For example, for a storage time interval of 6 months, all the forecasted time windows older than 6 months would be automatically deleted. This feature is implemented to avoid saturation on the storage, however, is the responsibility of the user to define this storage time interval taking into account the granularity used for the predicted workload.

Performance profiles service connector

This component is responsible for the connection to the PAE component that provides the performance profile of an application as well as information about the booting times of the virtual machines where the application is deployed. First, it validates if the required profiles are already stored in the database, in the case they are not it fetches all the available information about the application and the booting times for VMs. In addition, in case that in the middle of the derivation process some information is missing, for example, the Maximum Service Capacity (MSC) for a pod configuration that was not used before, then it performs a new HTTP request and updates the stored profiles.

Service Profiles database

Two different profiles are received from the Performance Profiles Service. The first one is the performance profile of the application and the second one refers to the information about booting and shutdown times of the different types of virtual machines. The data is stored in MongoDB using different collections per profile. It is stored permanently and updated either when is required in the policy derivation process or manually by the user.

Executor connector

This component is responsible for the interaction with the Execution Service of the PAE. The Execution Service is able to adapt the resources assigned to an application, therefore the executor connector in SPDT performs an HTTP request when the following actions are needed as part of the policy derivation process:

- **Fetch current state**: Since the initial configuration deployment influences the next scaling actions, the current application state is fetched from the Executor before starting the policy derivation process.
- **Schedule/Unschedule scaling actions**: A request is sent to the Execution service in order to schedule a new state per scaling action. In addition, when a scaling

policy is updated the previously planned scaling actions are unscheduled by performing a request that invalidates all the states that were scheduled after a certain timestamp.

Planner

This component uses the information retrieved from the other components along with the scaling strategies presented in the previous chapter and derives a scaling policy.

- **Planning how**: By default, all the scaling strategies are used to derive a list of candidate policies and select the cheapest afterward. However, the user can specify in the configuration file the name of the preferred scaling strategy to use.
- **Planning when**: The exact time *t* at which a scaling action should start is computed taking into account the resources estimated according to the scaling strategy applied, booting time of VMs that should be added, shutdown time of VMs that should be terminated and the pods' booting time.

Policies Database

All the policies derived for a time window are stored permanently in a database using MongoDB, regardless of whether the scaling actions were scheduled or not. In addition, besides the scaling actions that constitute the policy, it is stored all the information related to it. For example, policy id, scaling strategy used for the derivation, timestamps of when the policy derivation started and when it finished, cost, average transition time between states, VM types used, etc. A detailed list of the derived evaluation metrics is described in the next chapter.

Management

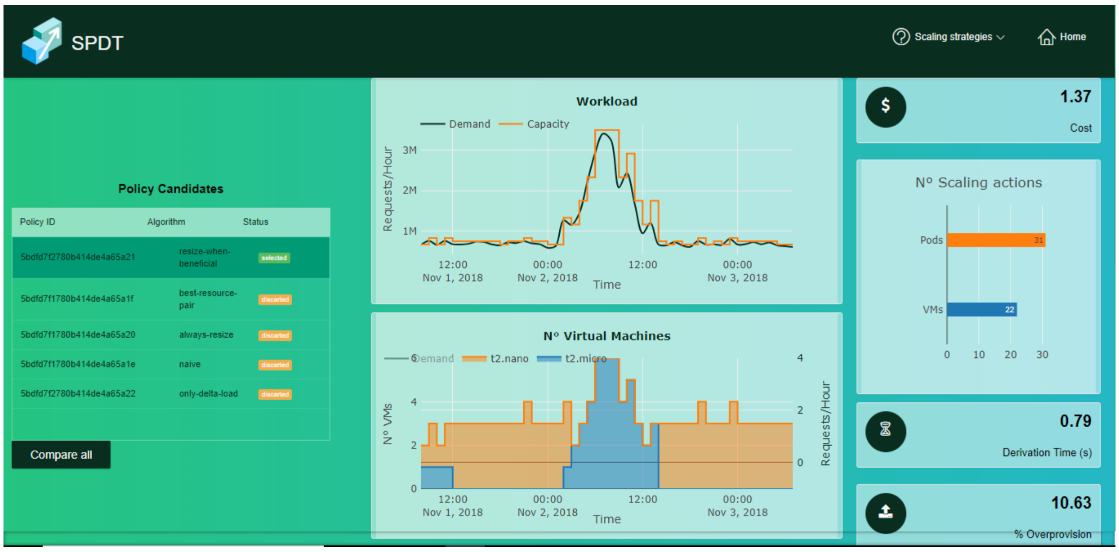
A Command Line Interface is implemented in order to manage some of the operations available on the tool such as download, invalidation, and deletion of policies.

Load Update Handler

In case the predicted number of requests in a time window for which a policy was already derived present significant changes, the Forecast component sends the information of the updated workload in order to update the policy. To handle this unexpected update a permanent listener to a port is provided so that updates can be received at any time. When an update arrives, the incoming data is validated to find out whether a new policy should be derived for the corresponding time window. In case it is necessary, a new policy derivation process is started, the old scaling actions are removed from the database and a request is sent to the Executor to invalidate the old states and schedule the new ones.

Dashboard

A dashboard is included to make easier for the user the understanding of how the resources change after the execution of a scaling action. In addition to the visualization of the application state throughout the time window, the metrics derived for the policy are shown. Figure 5.3 shows an example of how it looks like.



Policy Candidates

Policy ID	Algorithm	Status
5bdfd7f2780b414de4a65a21	resize-when-beneficial	selected
5bdfd7f1780b414de4a65a1f	best-resource-pair	discarted
5bdfd7f1780b414de4a65a20	always-resize	discarted
5bdfd7f1780b414de4a65a1e	naive	discarted
5bdfd7f2780b414de4a65a22	only-delta-load	discarted

Figure 5.3: Screenshoot of the dashboard

6 Experimental evaluation

The goal of the following experiments was to evaluate and compare the scaling policies derived by the scaling strategies described above. The evaluation considers different scenarios such as the type of application and workload, and it presents the results obtained for different evaluation metrics.

6.1 Experiment setup

Resources

The scaling strategies were evaluated using 5 of the instance types offered by Amazon EC2, a popular web service that provides secure, re-sizable compute capacity in the cloud. The Table 6.1 shows details of hardware configuration and cost per hour for the EC2 instance types. For clarity, the pricing model of billing per second was used during experiments.

Application types

In order to study the influence of the scaling strategies on different application profiles, 3 types of applications were considered: Database access, Compute intensive and Web access. Policies were derived for a representative application of each type showing how each one benefits differently from the resource limits assigned to the pods, e.g, for a pod replica with a configuration of CPU=0.5 and Memory=0.5Gb the web app is able to serve up to 670 requests/sec, the database app 49 requests/sec and the compute intensive application serve maximum 17 requests/sec.

Workload patterns

Scaling derivation strategies were tested using 3 patterns commonly present in cloud environments: Growing fast or increasing, unexpected burst and seasonal or cycle bursting. The workloads were different in volume and duration with 24, 48 and 192 hours length respectively.

VM Type	vCPU	Memory (Gb)	Price ($/Hour)
t2.nano	1	0.5	0.0058
t2.micro	1	1	0.0116
t2.small	1	2	0.023
t2.medium	2	4	0.0464
t2.large	2	8	0.0928

Table 6.1: VMs configurations and their prices

6.2 Evaluation metrics

- **Cost:** The cost of a policy is the aggregated cost of the resources billed by the Cloud Service Provider during the time window for which the policy is derived.
- **% Overprovision:** Overprovision is calculated based on the Maximum Service Capacity (MSC) provided at time t when MSC is higher than the number of requests demanded. The percentage of overprovisioning is the average overprovision throughout the time window for which the policy is derived.
- **% Underprovision:** Under provision is calculated based on the Maximum Service Capacity (MSC) provided at time t when MSC is lower than the number of requests demanded. The percentage of underprovisioning is the average underprovision throughout the time window for which the policy is derived.
- **Shadow time:** When migration of VM types is required during the transition between two states, two different deployment configurations have to overlap for some time in order to preserve the QoS. For example, to migrate from a t2.small VM to a t2.medium, the small VM is terminated only after the medium VM is spun up and ready to use. Therefore, shadow time is defined as the elapsed time during which the two configurations are running.
- **Derivation time:** This metric evaluates the efficiency of the strategy used to derive the scaling policy. It refers to the time that the strategy took to build the scaling plan. As described above each strategy uses different heuristics to take decisions, some use a simple approach to estimate resources, meanwhile, others use a more exhaustive search which could take longer to converge to a decision.
- **Average Transition time:** Is the time that takes the application to make the transition from an initial state to the desired state. It is calculated as the elapsed time from the time t when a scaling action starts until it finishes and all the resources are configured as defined by the desired state. The average transition

time is the average of the transition times between states considering all the scaling actions triggered throughout the time window.

- **Number of pods' scaling actions:** The number of scaling actions that require to scale pods either horizontally or vertically. From a general perspective, this is also the total number of scaling actions or state transitions throughout the time window.

- **Number of VM's scaling actions:** The number of scaling actions that required to scale virtual machines in order to support the scaling of pods.

6.3 Results analysis

Initial State: In this experiment, the resource configuration of the running application to scale consisted of 1 pod with resource limits of CPU=0.2 cores and Memory=0.2 Gb deployed on a virtual machine type t2.nano.

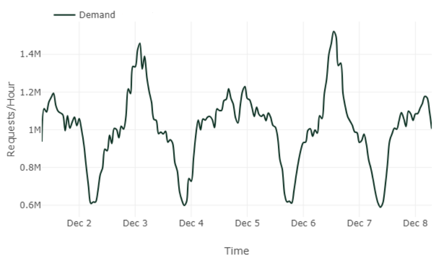
The figures 6.1-6.3 show the total cost of the scaling policies for a database application, web application and compute intensive application under different workload patterns. The following are observations after analyzing the results for the policies derived by the 5 scaling strategies:

- The total cost for policies presents a significant difference between applications. This is because each type of application needs a different amount of resources to meet the demanded load. For example, the web access application simply responds with a message when is called using REST request, therefore it has a good performance even with minimum resources. As is shown in the figures, most of the strategies have the same cost for the web application. This is explained given that for the tested workload, the application did not need to scale often. In addition, since the t2.nano is already the cheapest option to deploy an application, no further cost optimization was possible. In contrast, the compute intensive application demands a high number of scaling actions and resources to be able to meet the demanded QoS, and consequently is the most expensive application to scale.

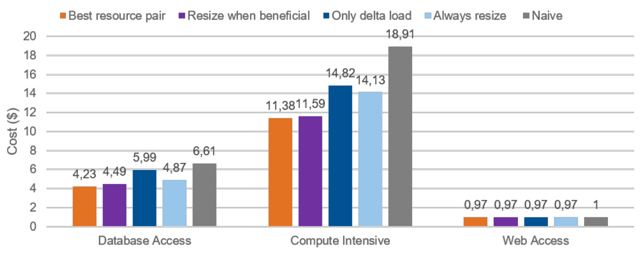
- For the tested application types, the *Best resource pair* strategy outperforms all other strategies by deriving a policy with the lowest total cost under the three tested workloads.

- The *Always resize* strategy is one of the most expensive for the database and compute intensive applications. This is explained given the overhead cost created by the frequent scaling actions that required migration from one type of virtual machine to another. In contrast, in the case of the web application few scaling actions are required and the time elapsed between them is long enough that the overhead cost due to migration can be mitigated.

- Comparison of the strategies that use vertical scaling for pods such as *Resize when beneficial*, against the strategies limited to use always horizontal scaling such as *Naive* or *Only delta load* shows that the total cost is reduced when the pods are resized at least once.
- In general, after testing for different workloads and using different configurations for the initial state, the scaling strategies could be ranked in the following order: 1. *Best resource pair*, 2.*Resize when beneficial*, 3.*Only delta load* 4.*Naive* 5.*Always resize*. This order is given when the focus is the cost of the derived policy, however for a complete evaluation more metrics should be considered in the analysis as proposed in the next section.

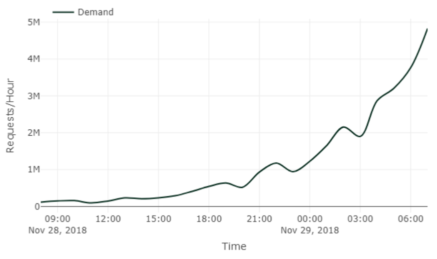


(a) Periodic bursting workload

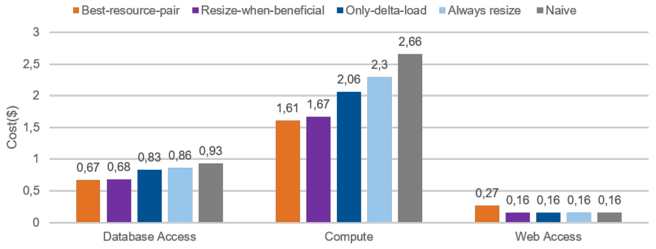


(b) Policies' total costs per scaling strategy

Figure 6.1: Experiment 1: Costs for a seasonal workload

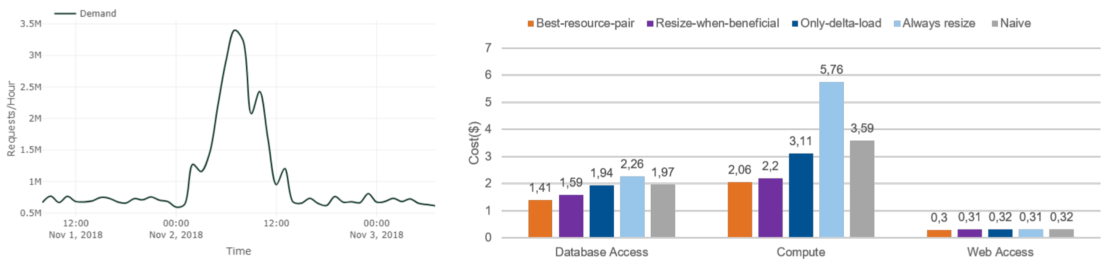


(a) Fast growing workload



(b) Policies' total costs per scaling strategy

Figure 6.2: Experiment 2: Costs for an increasing workload



(a) Aperiodic bursting workload (b) Policies' total costs per scaling strategy

Figure 6.3: Experiment 3: Costs for a bursting workload

6.4 Scaling plans comparison

To compare the scaling strategies considering only one metric could be biased, therefore this section shows the results of additional evaluation metrics derived for the case of scaling an application of type database access under an unexpected bursting workload. The table 6.2 shows the results of the evaluation metrics and figures 6.4 - 6.8 show the scaling policies derived for the different scaling strategies.

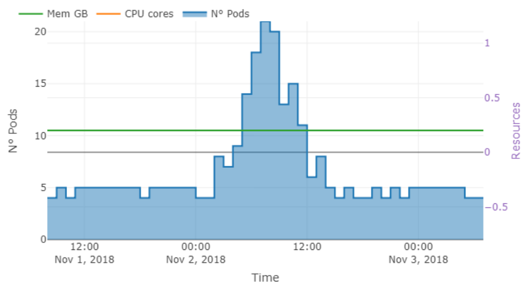
Metric	Naive	Best Resource Pair	Only Delta Load	Always Resize	Resize when beneficial
Cost ($)	1.97	1.41	1.94	2.26	1.59
Derivation time(s)	0.41	0.17	0.43	0.34	0.57
Avg transition time (s)	68.25	72.59	61.83	174.42	81.08
Avg shadow time	0	0	0	22.59	2.36
Number Scaling actions pods	29	12	29	29	25
Number Scaling actions VMs	29	12	24	28	24
% Over provisioning	7.48	26.55	7.48	13.57	13.52
% Under provisioning	0	0	0	4.52	1.89

Table 6.2: Evaluation metrics for a Database access application

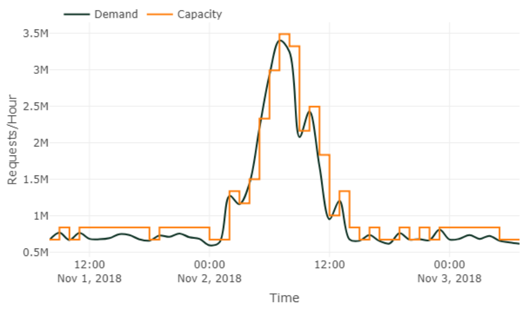
Each scaling policy presented in the figures below shows the pods and VM configurations at each time interval. The pod configurations include the number of replicas and the resource limits. In addition, subgraphs labeled with (b) show the comparison between the number of requests demanded and the capacity provided throughout the time window.

- **Naive strategy**

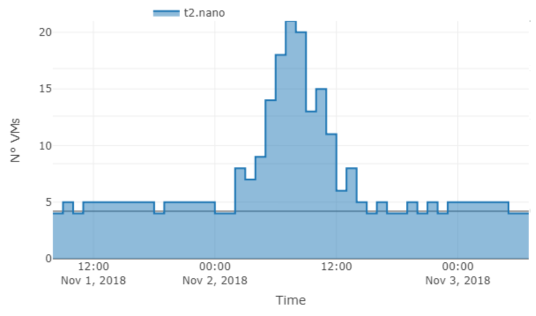
  The cost-efficiency of the *Naive* approach highly depends on the initial state, therefore the system administrator requires a good understanding of the application's performance to select the first configuration deployment. For example, in this test, the initial state uses a VM type t2.nano which even though is the cheapest option for a VM, using this strategy results in one of the most expensive policies. For this application, the use of t2.nano is not an optimal configuration as is suggested by other scaling strategies that use VMs type t2.micro instead. However, since this strategy only uses horizontal scaling, the transition time is in the top 2 of the fastest strategies to perform a scaling action.



(a) Pods' configuration



(b) Workload vs Capacity



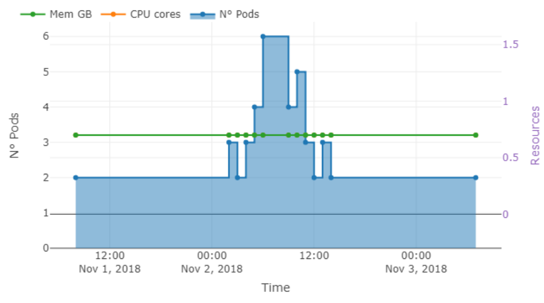
(c) VMs configuration

Figure 6.4: Scaling plan using Naive strategy

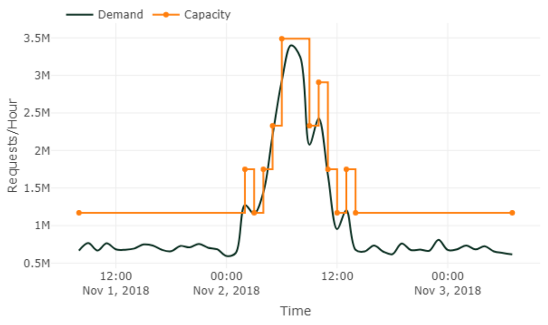
- **Best Resource Pair strategy**

  The first scaling action according to this strategy consists of changing the configuration of the initial state by increasing the pods' resource limits to CPU=0.7 cores Gb and Memory=0.7 Gb. Then, migrate the pods from a VM type t2.nano to a t2.micro. Thus, despite having a higher average percentage of overprovision in comparison to the Naive strategy, by resizing once pods and VMs this strategy derives the cheapest policy.

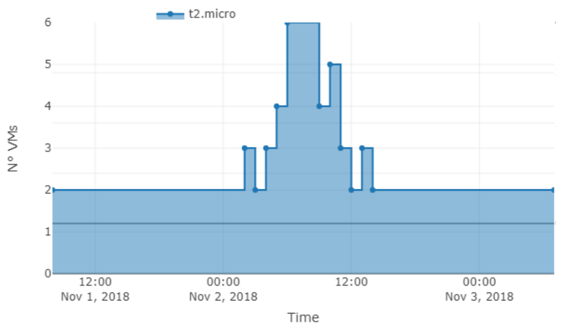
  The overhead cost caused by the migration from one VM type to another is easily mitigated since with the new configuration fewer scaling actions are performed afterward.



(a) Pods' configuration



(b) Workload vs Capacity



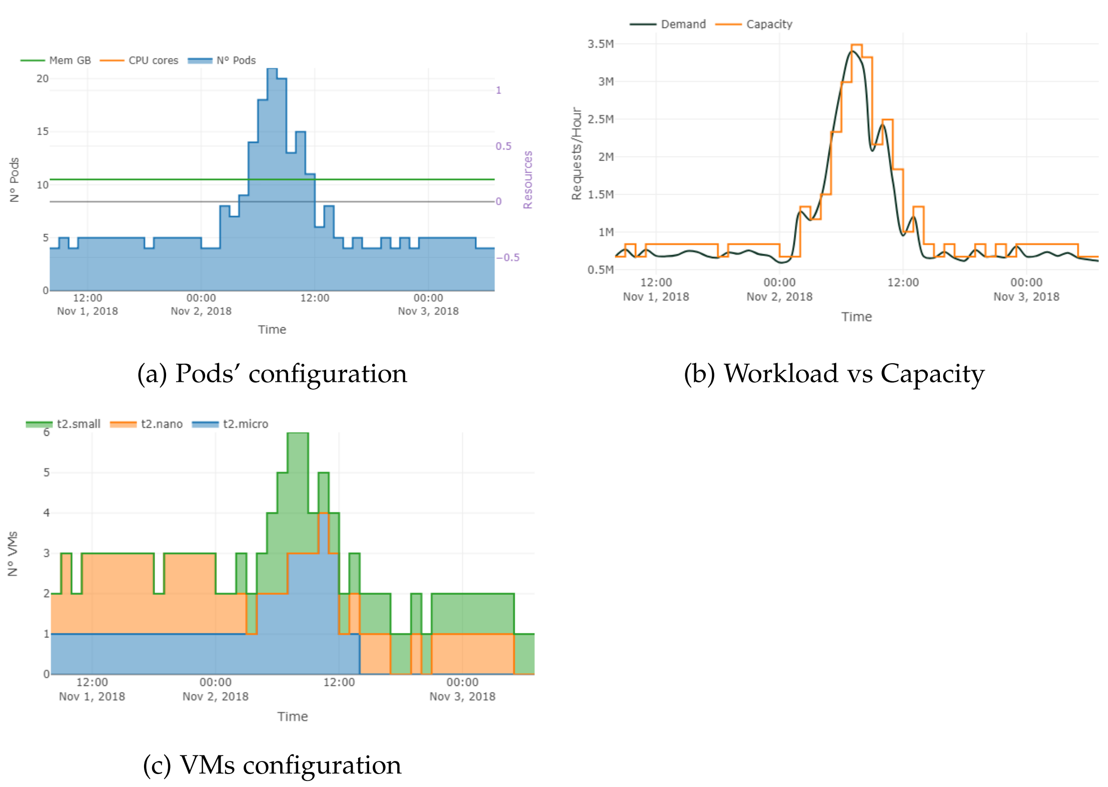
(c) VMs configuration

Figure 6.5: Scaling plan using Best Resource Pair strategy

- **Only Delta Load strategy**

  The policy derived using the *Only delta load* strategy has the shortest transition time. The scaling actions of this policy aim to scale only containers and it adds VMs of a size according to the increase of the load. Since the booting times of pods are faster than VMs, the transition between states is fast.

  In consequence, this strategy is ideal for applications that require quick adaptations to changes and the priority is to maintain the performance even by paying a higher cost. The figure 6.6c shows an example of the use of heterogeneous clusters to fulfill the demand. In this scenario, the VM type t2.nano is kept from the initial state and as more capacity is needed VMs type t2.micro are added. Then when a small increment is needed to reach the peak of the workload burst, VMs of type t2.small are added to the VM set and accordingly removed when the load decreases.



(a) Pods' configuration

(b) Workload vs Capacity

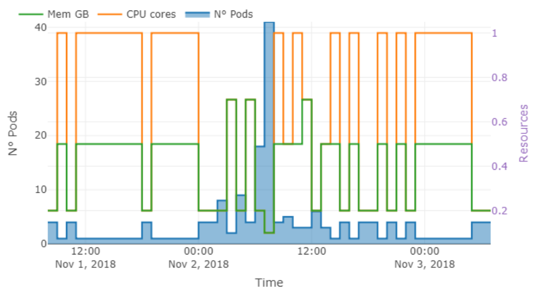
(c) VMs configuration

Figure 6.6: Scaling plan using Only Delta Load strategy

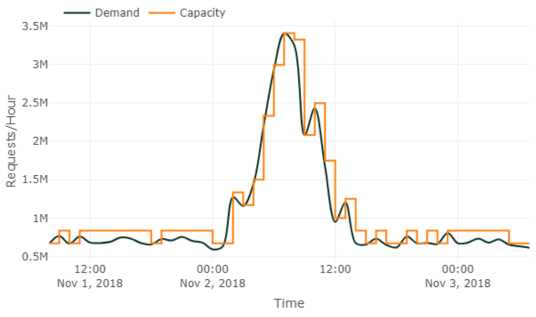
- **Always Resize strategy**

  As shown in figure 6.7a this strategy adjust the resource limits of pods according to the load. The orange line shows the CPU cores assigned to each pod at each time interval and in a similar manner, the green line shows the limit in resources for Memory in Gb.

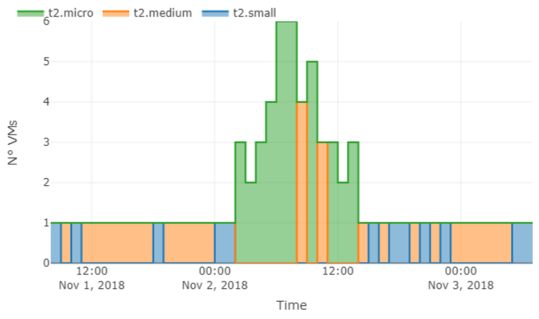
  For the scenario under test, this strategy suggests to increase the pod limits to CPU=1 and Memory=0.7 Gb. However, this configuration cannot be deployed on a t2.nano VM, therefore, migration to a t2.medium is required. Furthermore, the derivation time of this strategy is one of the highest. This could be explained given the search through all the available limit settings provided by the application profiles in order to find the new resource limits configuration for pods. Similarly, the transition time between states is the highest which is also reflected with the average of shadow time and is caused due to the number of migrations between VM types that need to be performed.



(a) Pods' configuration



(b) Workload vs Capacity



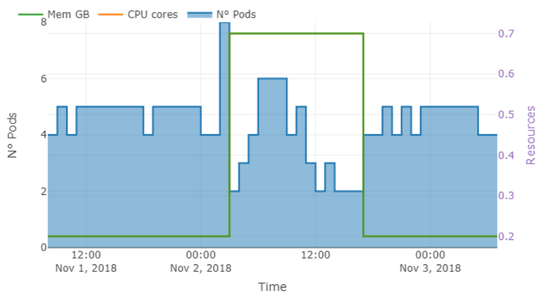
(c) VMs configuration

Figure 6.7: Scaling plan using Always Resize strategy

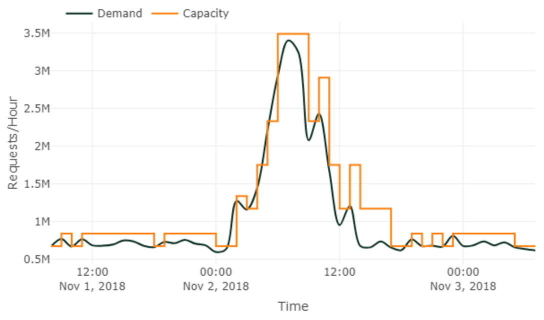
- **Resize when beneficial strategy**

  This strategy provides a policy with the second lowest cost. However, the derivation time of the strategy is one of the highest. This is explained because at each time interval a comparison to find an optimal VM set for delta load against an optimal VM set for the total load is performed.

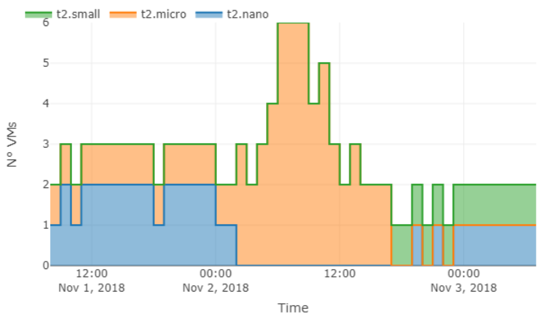
  As a result, the figure 6.8a shows how the pods are resized through the time window and the use of different VM types to fulfill the demand.



(a) Pods' configuration



(b) Workload vs Capacity



(c) VMs configuration

Figure 6.8: Scaling plan using Resize When Beneficial strategy

7 Conclusions

Cloud computing has enabled users to access computing services by using a pricing model per demand. This allows us to acquire resources when an application requires more capacity and release them when they are no longer needed. Although this is already an improvement in order to save costs, there are still several challenges at the moment of planning and estimating the right amount of resources to guarantee that the SLA is met without dramatically raising the operational cost.

This work presents a novel approach that takes into account different virtualization levels to derive scaling policies for cloud applications. The implementation of the SPDT uses 5 different scaling strategies named as: *Naive*, *Best Resource Pair*, *Only Delta Load*, *Always resize*, and *Resize when beneficial*; which based on the prediction of the workload and a profile of the application decide on the combination of resources for the microservice to be scaled.

Through experimental evaluations, it was shown that QoS requirements can be enforced by performing a cost-benefit analysis on the available resources. It has been proved that combining the benefits of vertical scaling and horizontal scaling at the container virtualization level decreases the costs of the resources used to serve an application. In contrast, vertical scaling at the level of virtual machines by replacing the VM type is only beneficial if the cost of migration can be mitigated by keeping the new deployment configuration for a period that lasts long enough, otherwise, the costs would rather increase significantly.

In addition, results demonstrated that the estimation of resources as a combination of virtual machines and containers is more beneficial than the estimation of each layer separately. Moreover, in the case of horizontal scaling, it was observed that heterogeneous clusters with more than two types of virtual machines do not guarantee an optimal solution but rather adds complexity to resource management.

Regarding the timing to take decisions, the experiments showed the advantages of using a proactive approach by including the prediction of the incoming workload in the analysis for the planning of a scaling action. Finally, the tests carried out using different workloads, also evidenced that different application types benefit differently from the available resources and therefore different scaling strategies can be applied.

The approaches and tool presented in the scope of this research have special relevance for cloud providers that have an impact on the revenue either because of penalties due to SLA violations, decrease in the user base due to a poor QoS or a significant cost overhead due to overprovisioning. Furthermore, the user can also benefit from the scaling policies derivation to plan the required budget for a future time window.

7.1 Limitations

Given that the implementation of this tool is still a prototype, it has some limitations that are worth to mention. Firstly, the developed tool does not consider the influence of the network bandwidth to take decisions. Secondly, the estimation of resources only considers the deployment configurations for containers and virtual machines, but it ignores additional resources such as storage volumes that could be necessary depending on the type of microservice. Additionally, the total cost is calculated based on the cost of the virtual infrastructure deployed with each derived policy, but it does not consider extra costs such as the cases where applications need licenses or billing charges based on burstable instances.

7.2 Summary of Contributions

This thesis makes a number of contributions to the problem of leasing resources to scale a cloud application under a dynamic workload. To begin, a characterization of the autoscaling process by describing the building blocks that should be considered when designing a scaling policy. According to this characterization and inspired on the techniques used by popular Cloud Service Providers to scale applications, a set of 5 scaling strategies based on heuristics that capture the user's understanding of the system were proposed. It is important to highlight that the proposed approaches include the coordinated elasticity of both Containers and Virtual Machines which according to the literature review is a topic that has not been deeply explored Then, these strategies were implemented in a tool named SPDT which is able to derive scaling policies for different types of applications.

The developed tool was integrated as a component of a Predictive Autoscaling Engine that carry out the whole autoscaling process. Next, an evaluation of the proposed scaling strategies was performed and the results provided insights about the scenarios when each scaling strategy is more suitable.

7.3 Future work

In the scope of a future research, the current work can be extended with modifications to the tool implementation as well as complementing the algorithms used in the scaling strategies. Following are some options for extension:

- **Scaling indicators:** Currently only the number of requests is used as a scaling indicator. Scaling indicators, especially from the cloud service provider, can be added with the hypothesis that a combination of both perspectives would provide more details to take the scaling decisions.
- **Policy selection metrics:** Even though different metrics are derived, currently the selection of the scaling policy is based only on cost. This can be improved by introducing an evaluation model that includes other metrics to select a policy and even allows the user to tune the trade-off of cost-efficiency.
- **Support for different CSP:** In the current implementation of the tool, the input information about Virtual Machines such as hardware configurations and prices is a responsibility of the system administrator who should provide the details in a JSON file. Although this is a straightforward approach, it can be enhanced by automatizing the connection to different Cloud Service Providers and extract the information from there.
- **Parallel policy derivation:** By default all scaling strategies are used to sequentially derive scaling policies, then the metrics are calculated and the cheapest option is selected. As an opportunity for improvement, the derivation of policies using the different strategies can be done in parallel to reduce the total derivation time.
- **Support for different pricing models:** From the variety of pricing models offered by the CSPs, at the moment only the on-demand model is supported. Therefore, an option for extension is to include the support for pricing models such as using spot instances or reserved.
- **Consider application structure:** Extension of the tool to derive policies for applications with different structures. Currently, only profiles for applications with one microservice are considered, however it would be interesting to include the scaling policies for applications composed of several microservices.

List of Figures

1.1 Overview of the Predictive Auto Scaling Engine 4

2.1 Autoscaling Process 9

3.1 Autoscaling policy building blocks 11

4.1 Code Snippet: JSON representation of a State 24
4.2 Strategy: Naive 27
4.3 Strategy: Best-Resource-Pair 28
4.4 Strategy: Only-Delta 30
4.5 Strategy: Alway-Resize 31
4.6 Strategy: Resize when beneficial 32

5.1 Information flow 36
5.2 Overall architecture of SPDT 37
5.3 Screenshoot of the dashboard 40

6.1 Experiment 1: Costs for a seasonal workload 44
6.2 Experiment 2: Costs for an increasing workload 44
6.3 Experiment 3: Costs for a bursting workload 45
6.4 Scaling plan using Naive strategy 46
6.5 Scaling plan using Best Resource Pair strategy 47
6.6 Scaling plan using Only Delta Load strategy 48
6.7 Scaling plan using Always Resize strategy 49
6.8 Scaling plan using Resize When Beneficial strategy 50

List of Tables

4.1 Notation used in equations 25
4.2 Summary scaling strategies 34

6.1 VMs configurations and their prices 42
6.2 Evaluation metrics for a Database access application 45

Bibliography

[1] K. H. J. D. G. C. Fox, *Distributed and cloud computing from parallel processing to the internet of things*. Morgan Kaufmann, 2013.

[2] R. B. C. V. S. Selvi, *Mastering cloud computing*. Morgan Kaufmann, 2013.

[3] A. Milenkoski, A. Iosup, S. Kounev, K. Sachs, P. Rygielski, J. Ding, W. Cirne, and F. Rosenberg, "Cloud usage patterns: A formalism for description of cloud usage scenarios spec rg cloud working group," 2013.

[4] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, "A review of auto-scaling techniques for elastic applications in cloud environments," *Journal of Grid Computing*, vol. 12, no. 4, pp. 559–592, 2014, ISSN: 15707873. DOI: 10.1007/s10723-014-9314-7.

[5] W. Fang, Z. Lu, J. Wu, and Z. Cao, "Rpps: A novel resource prediction and provisioning scheme in cloud data center," in *2012 IEEE Ninth International Conference on Services Computing*, Jun. 2012, pp. 609–616. DOI: 10.1109/SCC.2012.47.

[6] A. Jindal, V. Podolskiy, and M. Gerndt, "Multilayered cloud applications autoscaling performance estimation," in *2017 IEEE 7th International Symposium on Cloud and Service Computing (SC2)*, Nov. 2017, pp. 24–31. DOI: 10.1109/SC2.2017.12.

[7] M. A. S. Netto, C. Cardonha, R. L. F. Cunha, and M. D. Assuncao, "Evaluating auto-scaling strategies for cloud computing environments," in *2014 IEEE 22nd International Symposium on Modelling, Analysis Simulation of Computer and Telecommunication Systems*, Sep. 2014, pp. 187–196. DOI: 10.1109/MASCOTS.2014.32.

[8] R. Chen, S. Li, and Z. Li, "From monolith to microservices: A dataflow-driven approach," in *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, Dec. 2017, pp. 466–475. DOI: 10.1109/APSEC.2017.53.

[9] M. Villamizar, O. Garcés, L. Ochoa, H. Castro, L. Salamanca, M. Verano, R. Casallas, S. Gil, C. Valencia, A. Zambrano, and M. Lang, "Infrastructure cost comparison of running web applications in the cloud using aws lambda and monolithic and microservice architectures," in *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, May 2016, pp. 179–182.

[10] H. Khazaei, C. Barna, N. Beigi-Mohammadi, and M. Litoiu, "Efficiency analysis of provisioning microservices," in *2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, Dec. 2016, pp. 261–268. DOI: 10.1109/CloudCom.2016.0051.

[11] R. K. Barik, R. K. Lenka, K. R. Rao, and D. Ghose, "Performance analysis of virtual machines and containers in cloud computing," in *2016 International Conference on Computing, Communication and Automation (ICCCA)*, Apr. 2016, pp. 1204–1210. DOI: 10.1109/CCAA.2016.7813925.

[12] T. Ye, X. Guangtao, Q. Shiyou, and L. Minglu, "An auto-scaling framework for containerized elastic applications," in *2017 3rd International Conference on Big Data Computing and Communications (BIGCOM)*, vol. 00, Aug. 2018, pp. 422–430. DOI: 10.1109/BIGCOM.2017.40.

[13] W. Liao, S. Kuai, and Y. Leau, "Auto-scaling strategy for amazon web services in cloud computing," in *2015 IEEE International Conference on Smart City/SocialCom/-SustainCom (SmartCity)*, Dec. 2015, pp. 1059–1064. DOI: 10.1109/SmartCity.2015.209.

[14] H. Alipour and Y. Liu, "Online machine learning for cloud resource provisioning of microservice backend systems," in *2017 IEEE International Conference on Big Data (Big Data)*, Dec. 2017, pp. 2433–2441. DOI: 10.1109/BigData.2017.8258201.

[15] W. Fang, Z. Lu, J. Wu, and Z. Cao, "Rpps: A novel resource prediction and provisioning scheme in cloud data center," in *2012 IEEE Ninth International Conference on Services Computing*, Jun. 2012, pp. 609–616. DOI: 10.1109/SCC.2012.47.

[16] L. Lu, J. Yu, Y. Zhu, G. Xue, S. Qian, and M. Li, "Cost-efficient vm configuration algorithm in the cloud using mix scaling strategy," in *2017 IEEE International Conference on Communications (ICC)*, May 2017, pp. 1–6. DOI: 10.1109/ICC.2017.7997241.

[17] M. Sedaghat, F. Hernandez-Rodriguez, and E. Elmroth, "A virtual machine re-packing approach to the horizontal vs. vertical elasticity trade-off for cloud autoscaling," in *Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference*, ser. CAC '13, Miami, Florida, USA: ACM, 2013, 6:1–6:10, ISBN: 978-1-4503-2172-3. DOI: 10.1145/2494621.2494628.

[18] T. C. Chieu, A. Mohindra, A. A. Karve, and A. Segal, "Dynamic scaling of web applications in a virtualized cloud computing environment," in *2009 IEEE International Conference on e-Business Engineering*, Oct. 2009, pp. 281–286. DOI: 10.1109/ICEBE.2009.45.

[19] A. Ilyushkin, A. Ali-Eldin, N. Herbst, A. V. Papadopoulos, B. Ghit, D. Epema, and A. Iosup, "An experimental performance evaluation of autoscaling policies for complex workflows," in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ser. ICPE '17, L'Aquila, Italy: ACM, 2017, pp. 75–86, ISBN: 978-1-4503-4404-3. DOI: `10.1145/3030207.3030214`.

[20] H. Fernandez, G. Pierre, and T. Kielmann, "Autoscaling web applications in heterogeneous cloud infrastructures," in *2014 IEEE International Conference on Cloud Engineering*, Mar. 2014, pp. 195–204. DOI: `10.1109/IC2E.2014.25`.

[21] S. Taherizadeh and V. Stankovski, "Dynamic multi-level auto-scaling rules for containerized applications," *The Computer Journal*, bxy043, 2018. DOI: `10.1093/comjnl/bxy043`.

[22] Y. Hu, B. Deng, and F. Peng, "Autoscaling prediction models for cloud resource provisioning," in *2016 2nd IEEE International Conference on Computer and Communications (ICCC)*, Oct. 2016, pp. 1364–1369. DOI: `10.1109/CompComm.2016.7924927`.

[23] L. R. Moore, K. Bean, and T. Ellahi, "Transforming reactive auto-scaling into proactive auto-scaling," in *Proceedings of the 3rd International Workshop on Cloud Data and Platforms*, ser. CloudDP '13, Prague, Czech Republic: ACM, 2013, pp. 7–12, ISBN: 978-1-4503-2075-7. DOI: `10.1145/2460756.2460758`.

[24] ——, "A coordinated reactive and predictive approach to cloud elasticity," 2013.

[25] K. Hwang, Y. Shi, and X. Bai, "Scale-out vs. scale-up techniques for cloud performance and productivity," in *2014 IEEE 6th International Conference on Cloud Computing Technology and Science*, Dec. 2014, pp. 763–768. DOI: `10.1109/CloudCom.2014.66`.

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

[27] S. Dutta, S. Gera, A. Verma, and B. Viswanathan, "Smartscale: Automatic application scaling in enterprise clouds," in *2012 IEEE Fifth International Conference on Cloud Computing*, Jun. 2012, pp. 221–228. DOI: `10.1109/CLOUD.2012.12`.

[28] P. Desai, "A survey of performance comparison between virtual machines and containers," vol. 4, pp. 55–59, Jul. 2016.

[29] M. Tighe and M. Bauer, "Topology and application aware dynamic vm management in the cloud," *Journal of Grid Computing*, vol. 15, no. 2, pp. 273–294, Jun. 2017, ISSN: 1572-9184. DOI: `10.1007/s10723-017-9397-z`.

[30] F. Klinaku, M. Frank, and S. Becker, “Caus: An elasticity controller for a containerized microservice,” in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE ’18, Berlin, Germany: ACM, 2018, pp. 93–98, ISBN: 978-1-4503-5629-9. DOI: `10.1145/3185768.3186296`.

[31] Y. Al-Dhuraibi, F. Zalila, N. Djarallah, and P. Merle, “Coordinating vertical elasticity of both containers and virtual machines,” in *CLOSER*, 2018.

[32] C. Qu, R. N. Calheiros, and R. Buyya, “A reliable and cost-efficient auto-scaling system for web applications using heterogeneous spot instances,” *J. Netw. Comput. Appl.*, vol. 65, no. C, pp. 167–180, Apr. 2016, ISSN: 1084-8045. DOI: `10.1016/j.jnca.2016.03.001`.

[33] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle, “Autonomic vertical elasticity of docker containers with elasticdocker,” in *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*, Jun. 2017, pp. 472–479. DOI: `10.1109/CLOUD.2017.67`.

[34] H. Ghanbari, B. Simmons, M. Litoiu, and G. Iszlai, “Exploring alternative approaches to implement an elasticity policy,” in *2011 IEEE 4th International Conference on Cloud Computing*, Jul. 2011, pp. 716–723. DOI: `10.1109/CLOUD.2011.101`.

[35] R. Han, M. M. Ghanem, L. Guo, Y. Guo, and M. Osmond, “Enabling cost-aware and adaptive elasticity of multi-tier cloud applications,” *Future Gener. Comput. Syst.*, vol. 32, pp. 82–98, Mar. 2014, ISSN: 0167-739X. DOI: `10.1016/j.future.2012.05.018`.