EFFICIENT RESOURCE MANAGEMENT IN CLOUD DATA CENTERS

A MACHINE LEARNING APPROACH

TANIA LORIDO BOTRÁN

supervised by

SERGIO HUERTA LARA
BORJA SANZ URQUIJO

Bilbao, 2018

To my mum
and my grandparents

"If you can't be a pine on the top of the hill
be a scrub in the valley
–but be the best little scrub by the side of the rill;
be a bush if you can't be a tree.
Be the best of whatever you are!"
Douglas Malloch

Abstract

This thesis analyses the resource allocation problem in cloud data centers. Those typically run millions of applications in a virtualized environment (Virtual Machines or containers) that serves are the middleware layer between the physical servers and the user applications. Virtualization brings many benefits in resource sharing, with the easiness in creating and destroying instances, migrating VM from one physical machine to another, and great support for an "on-demand" computational model.

However, all these benefits do not come for free. They impose new challenges in resource sharing among different user applications. This dissertation studies the different problems associated with the resource management problem in cloud data centers, that are mainly: (1) Virtual Machine placement, (2) application auto-scaling, and (3) interference detection.

The resources offered by physical servers, organized in several data centers, are provided in the form of abstract compute units that are implemented as Virtual Machines (VMs). Each VM is assigned a pre-configured set of resources, including: number of cores, amount of memory, disk and network bandwidth.Virtualized data centers support a large variety of applications, including batch jobs (typically used for scientific applications), and web applications (e.g. an online bookshop). Each application is deployed on a set of VMs, which can be allocated to any collection of physical servers in the data center. The problem of assigning a physical location to each VM is known as *VM placement* and it is performed by the manager of the cloud infrastructure.

Cloud computing environments offer the user the capability of running their applications in an elastic manner, using only the resources they need, and paying for what they use. However, to take advantage of this flexibility, it is advisable to use an auto-scaling technique that adjusts the resources to the incoming workload, both reducing the overall cost and complying with the Service Level Objective. We propose a taxonomy consisting of five categories to classify state-of-art auto-scaling techniques: static threshold-based rules, time series analysis, control theory, reinforcement learning and queuing theory. Furthermore, we present a comparison of some auto-scaling techniques (both reactive and proactive) proposed in the literature, plus a new approaches based on rules with dynamic thresholds. Results show that dynamic thresholds avoid the bad performance derived from a bad threshold selection.

VMs or containers with different resource usage needs may be co-located in the same physical machine. Resource sharing (including CPU, memory or cache) may cause resource contention bottleneck, i.e., two VMs (or group of containers) compete for the same resources, but the resource capacity is not enough for both of them. This leads to *anomalies* in the resource usage of the application

that may penalize application performance. The noisy neighbor problem occurs when an application takes away the resources assigned to another one. We propose a lightweight unsupervised algorithm that is able to effectively detect these anomalies on different types of applications.

Our detection algorithm is based on clustering comparison and uses some novel distance metrics. The final contribution of this thesis is a set of new distance metrics that target the comparison of hard clusterings.

Contents

1 Introduction **3**
1.1 Popularity of Cloud computing 4
1.2 Cloud computing applications 5
1.3 Social impact of cloud computing 6
1.4 Goals of a Cloud provider 7
1.5 Challenges of Resource Management in Data Centers 8
1.6 Thesis Hypothesis and Objectives 9
1.7 Thesis Organization 10

2 Background **13**
2.1 What is Cloud Computing? 13
2.1.1 Characteristics of cloud computing 13
2.1.2 Cloud Computing service models 13
2.1.3 Cloud Computing deployment models 14
2.1.4 Virtual machines vs Containers 14
2.2 Virtual Machine Placement Problem 15
2.2.1 State-of-art in VM placement 16
2.3 Elasticity and Auto-scaling Process 17
2.3.1 Elastic Application 17
2.3.2 Auto-scaling Process: The MAPE loop 18
2.4 Noisy-neighbor effect and Performance Anomaly Detection 21
2.4.1 Noisy Neighbor Problem 21
2.4.2 General strategies to address it 22
2.4.3 State-of-art 23

3 Modeling the Cloud Data Center **29**
3.1 Modeling Applications 30
3.2 Modeling Application Workloads 32
3.3 Describing the Data Center Structure 34

3.4 Modeling Power Requirements 36

4 Application Placement: An Optimization approach 39

4.1 Multi-objective Optimization Algorithms 41
4.1.1 Why multi-objective approach? 41
4.1.2 Algorithms 41
4.2 Topology-aware Optimization 43
4.2.1 Problem Definition 43
4.2.2 Problem-specific Operators 44
4.3 Experimental Framework 45
4.3.1 Simulation Environment 46
4.3.2 Experiments to Compare Optimization Algorithms 46
4.3.3 Experiments to Evaluate VM Placement in Realistic Scenarios 47
4.4 Analysis of Results 48
4.4.1 Simplified Experiments: Evaluation of Optimization Algorithms 48
4.4.2 Detailed Experiments: Evaluation of the Benefits of Optimization-based Placement 49
4.5 Summary and Conclusions 49

5 Taxonomy on Auto-Scaling Techniques 53

5.1 Auto-scaling Taxonomy 54
5.2 Threshold-based Rules 57
5.2.1 Description of the Technique 57
5.2.2 Review of Proposals 58
5.3 Reinforcement Learning 61
5.3.1 Description of the Technique 61
5.3.2 Review of Proposals 63
5.4 Queuing Theory 66
5.4.1 Description of the Technique 67
5.4.2 Review of Proposals 69
5.5 Control Theory 71
5.5.1 Description of the Technique 71
5.5.2 Review of Proposals 73
5.6 Time Series Analysis 75
5.6.1 Description of the Technique 76
5.6.2 Review of Proposals 80
5.7 Summary and Conclusions 83

6 Dynamic threshold rules for Auto-scaling 87
6.1 Selected Auto-scaling Techniques 88
6.2 Experiments 90
6.2.1 Impact of load balancing policy 91
6.2.2 Reactive techniques 91
6.2.3 Proactive techniques 92
6.2.4 Comparing Reactive vs. Proactive techniques 93
6.3 Summary and Conclusions 93

7 An online algorithm to detect the Noisy Neighbor Problem 99
7.1 Our requirements 99
7.2 Anomaly Detection Algorithm 100
7.2.1 Scenario and System Architecture 100
7.2.2 Challenges 101
7.2.3 General Strategy 101
7.2.4 Techniques 103
7.2.5 Selected Techniques 106
7.2.6 Algorithm Description 107
7.2.7 Parameters and constraints 108
7.3 Experimental Framework 109
7.3.1 Evaluation 109
7.3.2 Application Benchmarks 109
7.3.3 Testbed 111
7.3.4 Parameter Selection 111
7.4 Analysis of Results 112
7.4.1 Computational requirements of our proposed algorithm . . 112
7.4.2 Parameter Tuning 117
7.4.3 Algorithm Evaluation 117
7.4.4 Comparison against other methods 120
7.4.5 Discussion of results 120
7.4.6 Advantages and disadvantages of our proposal 123

8 Spatially-Aware Distances for Hard Clusterings 125
8.1 Introduction 125
8.2 Background and Related Work 126
8.3 Formal Definition of Distance Measures 128
8.4 Distance Measure Validation 131
8.4.1 Minimal Properties for Metric Definition 132
8.5 Formal Validation of Metric Properties 132
8.5.1 Positivity and Symmetry 132
8.5.2 Identity of indiscernibles 133

8.5.3 Desirable Properties for Distance Measures 135
8.5.4 Comparison against other measures 139
8.5.5 Experiments with real datasets 140
8.6 Conclusions and Summary 142

9 Summary and Conclusions 145
9.1 Conclusions 145
9.2 Future Research Directions 146
9.2.1 Application placement 146
9.2.2 Auto-scaling 147
9.2.3 Anomaly detection 148
9.2.4 Comparison metrics for clusterings 148

Publications

Journal Papers

- Tania Lorido-Botran, Sergio Huerta, Luis Tomas, Johan Tordsson, and Borja Sanz. An Unsupervised Approach to Online Noisy-Neighbor Detection in Cloud Data Centers. *Expert Systems With Applications*, 2017

- Tania Lorido-Botrán, Sergio Huerta, and Borja Sanz. Towards Spatially-Aware Comparison of Hard Clusterings. 2017

- Tania Lorido-Botran, Jose Miguel-Alonso, and Jose A. Lozano. A Review of Auto-scaling Techniques for Elastic Applications in Cloud Environments. *Journal of Grid Computing*, pages 1–34, 2014. ISSN 1570-7873. doi: 10.1007/s10723-014-9314-7

- Jose A. Pascual, Tania Lorido-Botran, Jose Miguel-Alonso, and Jose A. Lozano. Towards a Greener Cloud Infrastructure Management using Optimized Placement Policies. *Journal of Grid Computing, Special Issue*, 2014. doi: 10.1007/s10723-014-9312-9

Conference and Workshop Papers

- Tania Lorido-Botran, Jose A. Pascual, Jose Miguel-Alonso, and Jose A. Lozano. Optimization of Application Placement towards a Greener Cloud Infrastructure. In *EvoPar*, 2014

- Tania Lorido-Botran, Jose Miguel-Alonso, and Jose Antonio Lozano. Comparison of Auto-scaling Techniques for Cloud Environments. In *Actas de las XXIV Jornadas de Paralelismo (SARTECO), CEDI*, 2013

Chapter 1

Introduction

The origin of Cloud Computing can go as far as 1960, when John McCarthy predicted that *computation may be organized as a public utility* like electricity and water [7]. Much later, in late 1990s, Grid Computing systems become popular and provided flexible environments to run batch jobs (scientific computing). Grid Computing was the first real implementation of computation as a utility.

With the usage of virtualization, Cloud Computing slowly replaced Grid Computing. One might ask: What is exactly *Cloud Computing*? In this dissertation we assume that is a business model built on top of data centers. Cloud Computing is becoming increasingly popular because, among other advantages, allows customers to easily deploy elastic applications, greatly simplifying the process of acquiring and releasing resources to a running application, while paying only for the resources actually allocated (pay-per-use or pay-as-you-go model). Elasticity and dynamism are two key concepts of cloud computing.

What enables this elasticity? A data center is simply a bunch of physical servers connected by some good network infrastructure (usually tree-tiered). Virtualization is used to provide resources in the form of Virtual Machines (VMs) or containers. Virtualization enables the execution of different application in the same physical servers, minimizing their interferences. It is possible to start and stop a VM or container in seconds (or minutes) and even migrate it to another physical server. This is the key to the *Cloud Data Center*, to the flexibility that enables the Cloud computing as business model.

Three main markets are associated to cloud computing:

Infrastructure-as-a-Service (IaaS) designates the provision of information technology and network resources such as processing, storage and bandwidth as well as management middleware. Examples are Amazon EC2 [8], RackSpace [9] and Google Compute Engine [10].

Platform-as-a-Service (PaaS) designates programming environments and tools hosted and supported by cloud providers that can be used by consumers to build and deploy applications onto the cloud infrastructure. Examples include Amazon Elastic Beanstalk [11], Google App Engine [12], and Microsoft Windows Azure [13].

Software-as-a-Service (SaaS) designates hosted vendor applications. For example, Google Apps [14], Microsoft Office 365 [15] and Salesforce.com [16].

A typical scenario could be a company that wants to host an application, and for this purpose leases resources from a IaaS provider such as Amazon EC2. From now on the following terms will be used:

Provider. It refers mainly to the IaaS provider, that offers virtually *unlimited* resources in the form of VMs. A PaaS provider may also play this role.

Client. The client is the customer of the IaaS or PaaS service, that uses it for hosting the application. In other words, it is the application owner.

User. This is the end user that accesses the application and generates the workload or demand that drives its behavior.

Resource scaling can be either horizontal or vertical. In horizontal scaling (*scaling out/in*), the resource unit is the server replica (running on a VM), and new replicas are added or released as needed. In contrast, vertical scaling (*scaling up/down*) consists of changing the resources assigned to an already running VM, for example, increasing (or reducing) the allocated CPU power or the memory. Most common operating systems do not allow on-the-fly (without rebooting) changes on the machine on which it runs (even if it is a VM); for this reason, most cloud providers only offer horizontal scaling.

1.1 Popularity of Cloud computing

Cloud computing has revolutionized the Information and Communication Technology (ICT) industry by enabling on-demand provisioning of elastic computing resources on a pay-as-you-go basis. The increased popularity has cause a rapid

growth of customer demands for processing power, storage and communication. A revealing example comes with Google: their data centers process an average of 40 million searches per second, resulting in 3.5 billion searches per day and 1.2 trillion searches per year, Internet Live Stats reports. That's up from 795.2 million searcher per year in 1999, one year after Google was launched [17].

The prospective trend for cloud computing is to keep increasing over time. A report by Cisco Systems [18] predicts that global data center IP traffic will grow 3-fold over the next 5 years, and overall data center workloads and compute instances will be more than double (2.3-fold) from 2016 to 2021. It also predicts that the data stored in data centers will nearly quintuple by 2021 to reach to reach 1.3 ZB.

In order to cope with the rapid growth of demand, Cloud providers such as Amazon, Google and Microsoft are deploying large-scale data centers around the globes. Recent report shows that Amazon operates at least 30 data centers in its global network, each comprising 50,000 to 80,000 servers [19]. The number of servers per data center is also increasing. Technical presentations by Facebook staff suggested that as of June 2010 the company was running at least 60,000 servers in its data centers, up from 30,000 in 2009 and 10,000 back in April 2008. Today it is estimated that each Facebook data center might have hundreds of thousands[20].

As expected, this leads to huge investments from bigger cloud providers in data centers. At the Google Cloud Next conference in San Francisco in March 2017, the company revealed that it spent nearly $30 billion on data centers over the preceding three years.[17]. Facebook reported that it owned about $3.63 billion in “network equipment” as of the end of 2015 — up from $3.02 billion in 2014 [20].

The revenue is a good measure of cloud computing *health*. Microsoft reported in 2018 a 98% increase in its revenue [21], closely followed by competitors Amazon and IBM. This is a good sign that Cloud Computing will stay with us for a long time.

1.2 Cloud computing applications

Cloud computing has changed the IT industry by enabling on-demand provisioning of elastic computing resources on a pay-as-you-go basis. Organization have two main options, either outsource their computational needs to a third-party or build their own private data center. Outsourcing allows the organization high up-front investments in a private computing infrastructure and consequent costs of maintenance and upgrades. This scenario is particulary appealing for start-ups, as it provides greater flexibility and more control over the costs. As the business grows, cloud providers offer *infinite* resources to deploy new services. In the case

that the company fails, there is no loss in material assets, as the start-up can stop renting resources from the cloud provider at any time. According to[22], companies can use only 60-80% of the budget that is used to run IT and convert fixed IT costs into variable spends. Bigger and well-stablished organizations might prefer to set up their own private data center and benefit from improved resource management schemes.

But what is the applicability of Cloud Computing? It can be applied to many industries, including education, health and software developments. Here we enumerate some of the most popular areas of applicability [23]:

- Proof of concept and product development. Cloud offers an ideal scenario to test a new idea that can be easily discarded. It is also suitable to create a testbed that supports the whole cycle of product developtment, and separate environments for testing or production

- Backup and disaster recovery: Data centers are spread accross different continents. This is the highest level of security and protection in case of natural disasters.

- Traffic bursting: A retail business owning its own data center might suffer from spikes in demand during certain periods of the year (e.g. Black Friday). Cloud providers offer a great solution for this sporadic ocasions.

- Big data analytics and scientific computing: Big Data requires is very computationally demanding and also requires inmense amount of storage to handle huge datasets. Cloud computing is becoming a popular solution of Big Data tasks and scientific computing. Some public cloud vendors even offer specialized services focuses on data processing.

- Web hosting, file storage: The Cloud can also be used for tradicional needs like web hosting and file storage.

1.3 Social impact of cloud computing

As IBM states [24], Cloud computing is changing our lives in many ways, even without us noticing. The world has become a global place and any piece of news might become viral. Cloud computing has changed the way we interact (e.g. think about the likes on Facebook or Youtube), or our political preferences. It is present in every day of our lifes, with both positive and negative effects.

We have already covered the positive effects of cloud technology. First, it benefits business by allowing them to grow and cut down on costs, and also providing coverage to multiple tasks from web hosting to Big Data Analytics.

Cloud technology is important for users, by keeping them close to a variety of data and applications that are readily available online. It is worth to mention the value of Cloud computing for social good: it offers other benefits to developing countries since they no longer have the burden of investing in costly infrastructures and get to into data and applications that are readily available in the cloud.

A great example is an startup called MobileWorks [22] that has developed a mobile web application to allow low income device users in seven countries execute simple data entry tasks, web research, content creation and more on a project basis to supplement their income. Thanks for the availability of free cloud computing services, the application was launched with zero capital, allowing them to deploy the software in developing nations.

However, the impact of cloud computing in our society is not only a fairy tale story. It has created a great dependability on cloud services (third-parties). This dependability manifests in different ways. A business relies on a cloud provider guaranteeing a certain QoS level, but if it fails to do so, the business may suffer important finantial losses. Depending on the service provided, it can even disable the company's main activity or have disastreous consequences. Let us imagine the case of self-driving cars relying on cloud systems to compute the distance to pedestrians: the slighest error could cause the loss of a life.

Another direct problem derived from sharing our data with a third-party is the lack of security and privacy [25]. Who can access our data? What if someone uses our bank account details to commit fraud or a crime?

1.4 Goals of a Cloud provider

A cloud provider is responsible for running millions of applications across multiple data centers. We can identify two main goals from the perspective of the cloud provider, that are: first, to maximize revenue and, second, to guarantee a certain Quality of Service (QoS). Desirably, the resource management strategy for data centers should find a trade-off between both objectives. However, resource management is not that straight-forward, and cloud providers need to deal with a series of problems, such as the unpredictability of the load, the under-utilization of real data centers and the high maintenance cost of each data center.

Incoming requests to cloud applications can be really bursty and vary according to external events, like the characteristics of the workload volume or the workload type mix. A typical example was the traffic surge arised from Michael Jackson's death, which resulted in disruptions in several services [26]. The main consequence of a service disruption is a violation of an SLA (Service Level Agreement) between the cloud provider and the application owner. The cloud provider has to face significant finantial losses for not guaranteeing a *good* QoS.

Maximizing revenue immediately translates to using the total of available resources. However, several studies show the worrying underutilization of cloud data centers. [27] analyze a 12000-nodes Google clusters and find out a low CPU utilization of only 25%-35%, together with an also low 40% of memory usage. Similarly, a production cluster at Twitter with a thousand nodes reports CPU utilization below 20% and memory usage at 40%-50% [28]. The same study estimates that the overall utilization of dedicated Cloud facilities is even worse, between 6%-12%.

Finally, there is a general concern about the increased energy consumption required by data centers. An average data center consumes as much energy as 25,000 households, as reported by [29]. This energy consumption results in increased Total Cost of Ownership of the data center, and less revenue for the cloud provider. Microsoft has tried to address this problem by creating *submarine* data centers, that will use the currents to generate electricity [30] and the cooldown thanks to low water temperatures. Although appealing, we can see obvious inconvenients like the maintenance of such data centers.

There is enough evidence that proves that data centers are not managed in the best way. Existing strategies for resource management are insufficient. Next section will explore the main challenges towards an efficient way to use existing resources.

1.5 Challenges of Resource Management in Data Centers

As we said previously, Cloud Computing has two very appealing characteristics for users, elasticity and a pay-as-you-go scheme: get more resources whenever you want, release them as soon as you do not need them, and on top of that, only pay for what you need. However, the whole scheme brings new challenges to the owner of the cloud data center. How to efficiently manage the resources that change dynamically over time? We can identify three main challenges associated to the Resource Management of a Cloud Data Center: application placement, auto-scaling and performance anomaly detection.

Applications hosted in cloud environments can be of very diverse nature: batch jobs, map-reduce tasks, massively multiplayer online role-playing games (MMORPGs), web applications, video streaming services, and a large etcetera. An IaaS provider has to decide the best placement of each application within the data center(s). Better said, the best mapping between a set of VMs or containers that compose each application. This is usually referred to as *Virtual Machine (or application) placement problem.* Overall, the IaaS provider has to deal with you main goals, maximize the resource usage of the data center and also guarantee

a minimal quality service. This is usually denoted as SLA, and it is a contract between the client and the provider.

If the problem was not complex enough, the number of VMs or containers for an application may vary over-time. It might start a new VM if there is a spike in the incoming workload or stop several VMs that have been idle for a certain period of time. This tool is referred as the *auto-scaler* and its goal is to allow applications to acquire and release resources dynamically, adjusting to changing demands. Auto-scaling in itself is a pretty challenging task, given the uncertainty of the incoming workload.

Virtualization enables running several application on top of the same physical server and sharing the same resources. Each application will generally underuse the resources assigned to it, so an IaaS provider will usually overbook the resources, that is, pack as many applications as possible in the same physical server. However, there is always a risk of violating an SLA. This scenario is very prone to have interferences among VMs or containers. A VM might use 100% of a CPU core assigned to it, leaving the neighbor VM without any resources for a while. This is a very common problem denoted as *noisy-neighbor problem.* An optimal placement is never enough. We need to provide a good anomaly detection method to prevent performance issues.

1.6 Thesis Hypothesis and Objectives

The focus of this thesis is to find a new strategy for managing resources in cloud computing infrastructure. We propose the following hypothesis:

By the development of new virtual machine placement, auto-scaling and prediction techniques, we can optimize the resource usage in cloud environments, both from the users' and the infrastructure perspective.

In order to accomplish the stated hypothesis, we enunciate a set of four objectives formulated as questions:

- How to map applications (a set of Virtual Machines) to available physical resources taking into account the energy usage and network utilization? Data centers run very heterogeneous workloads. It is necessary to find the best mapping for each application, and desirably, reduce the energy comsumption of the data center.
- How to optimally perform on-demand request of resources for an application? One of the biggest challenges in data centers involves reducing the under

utilization of resources. Applications are typically assigned many Virtual Machines that remain idle. A good automatic tool would be useful to adjust the resources assigned to an application based on the demand, while freeing unused resources.

- How to guarantee a minimum SLA in a shared environment prone to performance interferences? One of the challeges for the cloud provider is to maintain a good Quality of Service, by doing a good distribution of the resources among applications. However, this scenario is very prone to performance anomalies. Cloud providers need to detect these anomalies as soon as possible.

- How can we characterize/model the whole data center in order to test new resource management approaches? The data center is a huge entity, composed of physical resources (servers, topology infrastructure, cooling systems), and software components (applications, workloads). Testing new resource management approaches can be a lenghtly and very costly process. A good characterization of a data center would help researchers to test the efficiency of new ideas.

1.7 Thesis Organization

The present manuscript provides answers to all the objectives described in previous section and it is organized as follows:

- Chapter 2 presents the foundation of this thesis. It contains basic information about the environment, cloud data centers, and provides a description of the main challenges in managing virtualized resources, that is, Virtual Machine placement, application auto-scaling and performance anomaly detection.

- Chapter 3 proposes a set of models to characterize a data-center in full, including the network topology and energy consumption model, and a general application and workload modeling. This sets of model is used in next Chapter to test a VM placement solution

- Chapter 4 tackles the application placement problem, that is, mapping the VMs to the best set of physical servers. It proposes a novel multi-objective optimization approach to solve the initial mapping of an application to a set of cores, by taking into account the application communication pattern and the network topology. Besides, the solution reduces the energy consumption of the data center.

- Chapter 5 proposes a taxonomy for auto-scaling techniques and reviews the general state-of-art. Techniques can be classified into five categories that are: rules, control theory, queuing theory, reinforcement learning and time series. Although the review focuses on auto-scaling, many of these techniques are applicable to other resource management problems.

- Chapter 6 proposes a very easy to implement variant of rules with dynamic thresholds and compares against other state-of-art auto-scaling techniques.

- Chapter 7 tackles the noisy neighbor problem and proposes a lightweight detection algorithm to detect performance anomalies in the data center. It is application-agnostic solution and assumes no previous knowledge about past anomalies or the application's resource usage profile.

- The last Chapter, Chapter 8, (with a more technical nature) proposes a set of comparison metrics for hard clusterings. The anomaly detection algorithm is based on statistical distances to compare soft clusterings and inspired this contribution in the field of hard clusterings.

Chapter 2

Background

2.1 What is Cloud Computing?

We already said that cloud computing is a business model for a virtualized data center. A more formal definition is provided by NIST and states that *Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model is composed of five essential characteristics, three service models, and four deployment models.* [31]

2.1.1 Characteristics of cloud computing

Cloud computing can be categorized as a business model. The use of virtualization technologies enables the following characteristics:

- On-demand
- Pay-as-you-go: Only pay for the resources you need
- Rapid elasticity: Seamlessly scaling (acquiring and releasing resources)

2.1.2 Cloud Computing service models

Three main markets are associated to cloud computing:

Infrastructure-as-a-Service (IaaS) designates the provision of information technology and network resources such as processing, storage and bandwidth as well as management middleware. Examples are Amazon EC2 [3], RackSpace [20] and Google Compute Engine [13].

Platform-as-a-Service (PaaS) designates programming environments and tools hosted and supported by cloud providers that can be used by consumers to build and deploy applications onto the cloud infrastructure. Examples include Amazon Elastic Beanstalk [5], Google App Engine [10], and Microsoft Windows Azure [18].

Software-as-a-Service (SaaS) designates hosted vendor applications. For example, Google Apps [11], Microsoft Office 365 [17] and Salesforce.com [25].

2.1.3 Cloud Computing deployment models

- Private cloud. The cloud infrastructure is provisioned for exclusive use by a single organization comprising multiple consumers (e.g., business units). It may be owned, managed, and operated by the organization, a third party, or some combination of them, and it may exist on or off premises.
- Public cloud. The cloud infrastructure is provisioned for open use by the general public. It may be owned, managed, and operated by a business, academic, or government organization, or some combination of them. It exists on the premises of the cloud provider.
- Hybrid cloud. The cloud infrastructure is a composition of two or more distinct cloud infrastructures (private, community, or public) that remain unique entities, but are bound together by standardized or proprietary technology that enables data and application portability (e.g., cloud bursting for load balancing between clouds).

2.1.4 Virtual machines vs Containers

Virtualization is used as an intermediate layer to decouple applications from the data center infrastructure. It enables flexible allocation of physical resources to cloud applications. The amount of resources assigned to each application may be adjusted dynamically. It enables multi-tenancy, that is, multiple applications sharing the same physical server. Finally, virtualization simplifies the replication and scaling of applications.

There are two main technologies: hardware and operation system level virtualization. In the hardware-level virtualization, a hypervisor virtualizes physical

resources across several VMs. Each VM runs its own OS. Containers use OS virtualization, that is, resources are virtualized at OS level, by encapsulating Os processes and their dependencies.

Examples of VM management frameworks include vCenter, Openstack, and CloudStack. There are also container management frameworks such as Kubernetes and Docker Swarm.

For a long time, hardware virtualization has been the predominant technology in cloud data centers, but more recently containers are gaining increased popularity. They have less overhead than VMs; in contrast, VMs provide more robust isolation. [32] found that the performance of applications running on top of containers (LXC) are comparable to bare-metal OS. KVM shows different levels of overhead, depending on the resource, with up to 80% worse in disk throughput. In contrast, containers lack from isolation capabilities. According to [32], LXC containers are starved of resources against an CPU-adversarial workload, while the VM finishes with a 30% performance degradation.

An interested reader might wonder: *what is the best virtualization technology for Cloud Computing, containers or VMs?*. The answer depends on the ultimate needs of the user. In case that low overhead is a primary concern and the application might benefit from vertical scaling, then containers seems like the way to go. However, if the user want to ensure insolation in a multi-tenancy environment, it is advisable to use VMs at the virtualization technology. Most part of this thesis work could be applied to both containers and VMs, including auto-scaling, placement and interference detection.

2.2 Virtual Machine Placement Problem

Virtualized data centers support a large variety of applications, including batch jobs (typically used for scientific applications), and web applications (e.g. an online bookshop). Each application is deployed on a set of VMs, which can be allocated to any collection of physical servers in the data center. The problem of assigning a physical location to each VM is known as *VM placement* and it is performed by the manager of the cloud infrastructure. This manager is typically called the Infrastructure-as-a-Service (IaaS) provider.

The challenge for the provider is to host a large and diverse set of applications (VM sets from different clients) in the infrastructure trying to (1) maximize its revenue and (2) provide a good service to the clients. An adequate application placement would be able to maximize the resource usage of physical servers and reduce the energy consumption of the data center, for example, by turning off (or setting to idle state) the inactive servers and network elements (typically, switches). At the same time, the infrastructure management policies should trade

off the obtained revenue with the Quality of Service (QoS) agreed with the client, guaranteeing that each application receives the resources paid for.

2.2.1 State-of-art in VM placement

This thesis is focused on the problem of performing the initial placement of the collection of VMs that constitute a cloud application. We argue that making a *suitable* initial decision about VM placement is essential to keep the data center in a near-optimal state, both in terms of energy consumption and resource utilization, and also to reduce the future need of consolidation. Consequently, fewer VM migrations will be required, that directly implies less network overload due to transfers. This review of the literature pays special attention to papers that target the initial placement of applications.

Open-source tools for cloud management use rather simple placement policies. For example, Eucalyptus [33] implements FF and RR strategies that only consider the VM requirements and the availability of resources. It also implements a Power-Save policy that is similar to the ranking algorithm available in OpenNebula [34]: choosing first the most used servers (with room for the new demand) with the objective of minimizing the number of used servers and, therefore, the power consumption. Commercial tools for capacity planning, such as NetIQ PlateSpin Recon [35], VMware Capacity Planner [36] and IBM Workload Deployer [37] also focus on maximizing the resource usage and power consumption savings. None of these tools explain how VM placement is carried out.

Open-source tools do not consider the impact of network topology and the communication patterns of applications, but they have been analyzed in many research works [38] [39] [40] [41] [42] [43] [44] [45]. For example, authors in [41] propose grouping VMs and servers into clusters, addressing VM placement for each <VM-cluster, server-cluster> tuple as a Quadratic Assignment Problem (QAP). The VM clustering tries to maximize the intra-cluster communication and reduce the inter-cluster communication, but all VM-clusters are equal on size. The server set assigned to a VM-cluster is fixed. This work does not consider the energy consumed by physical servers. Authors in [40] follow a greedy heuristic approach, but they do not consider the large variety of applications that can run in the cloud. In [39] a greedy heuristic to improve the network utilization is presented, but it does not try to allocate the VMs in the minimum number of physical servers.

The energy consumption of a data center is derived from many elements including physical servers and the network infrastructure. However, most models for data center energy do not take into account the combined effects of these two elements. For example, authors in [41] do not consider the energy consumed by servers and [46] do not include the network infrastructure in its energy model.

Some authors address each the of the objectives separately (e.g. energy consumption and network usage) [47]. However, the VM placement problem can benefit from a multi-objective approach, where several objectives can be optimized at the same time. Tziritas et al. [46] compare a single vs. multi-objective optimization approach and state that the latter is able to find the best trade-off between the total energy consumption and the network overhead.

2.3 Elasticity and Auto-scaling Process

2.3.1 Elastic Application

An *elastic* application has the capability of being scaled (horizontally or vertically) to make it adjust to the input, variable workload. Applications of this class are normally based on a load balancer (a dispatcher) and a collection of identical servers. In a cluster environment, those would be physical servers but, in cloud environments, servers are hosted in VMs and, therefore, the terms servers and VMs can be used interchangeably. An auto-scaler is in charge of making decisions about scaling actions, without the intervention of a human manager.

Although several applications can be considered elastic (e.g. MMORPG or video streaming servers), most of the literature is focused on web applications. These typically include a business-logic tier, containing the application logic, and a persistence or data base tier. The literature pays most of its attention to the business-logic tier, that can be easily scaled. There are some works that study auto-scaling at the persistence tier, although replicating distributed databases brings out additional issues. Our description of auto-scaling techniques is as neutral as possible, without focusing on any particular application class or tier. However, given the literature bias towards web applications, it is unavoidable to see it reflected in this survey.

Let us consider an elastic application deployed over a pool of n VMs. VMs may have the same or different assigned amount of resources (e.g. 1GB of memory, 2 CPUs,...), but each VM has its own unique identifier (it could be its IP address). The requests received by the load balancer may come from real end users or from other application. We will assume that the execution time of a request may vary between milliseconds and minutes; long-running tasks lasting hours will not be considered in the auto-scaling problem, as they rather belong to a scheduling problem.

The load balancer will receive all the incoming requests and forward them to one of the servers in the pool. It may be implemented in different ways. These are a few examples: a specific, hardware/software combination (an Internet appliance), a part of the Domain Name System, a service offered by the IaaS provider, or even an ad-hoc part of the client's application. Whichever the chosen technology,

it is assumed that the load balancer has updated information about the VMs to use (the active ones): it will stop immediately sending requests to removed VMs, and it will start sending workload to newly added VMs. It is also assumed that each request will be assigned to a single VM, that will run it until completing the task associated to it. Several load balancing policies could be used, for example, random, round-robin or least-connection. In the case of heterogeneous collections of VMs, workload dispatching should be proportional to the processing power of the VMs. The revision of techniques to implement load balancers, distribute workload among VMs, control the admission of new requests, or redirect requests between VMs fall outside the scope of this survey.

2.3.2 Auto-scaling Process: The MAPE loop

The aim of the auto-scaler is to dynamically adapt the resources assigned to the elastic applications, depending on the input workload. This auto-scaler may be either an ad-hoc implementation for a particular application, or a generic service offered by the IaaS provider. In any case, the system should be able to find a trade-off between meeting the SLA of the application and minimizing the cost of renting cloud resources. Any auto-scaler faces several problems:

Under-provisioning: The application does not have enough resources to process all the incoming requests within the temporal limits imposed by the SLA. More resources are needed, but it takes some time from the moment they are requested until they are available. In case of sudden traffic bursts, an under-provisioned application may lead to many SLA violations, even to system congestion. It will take some time until the application returns to its normal operational state.

Over-provisioning: In this case the application has more resources than those needed to comply with the SLA. This is correct from the point of view of SLA, but the client is paying an unnecessary cost if the VMs are idle or lightly loaded. A certain degree of over-provisioning could be desirable in order to cope with small workload fluctuations. In general, neither a manual provision of resources, nor the one carried out by an auto-scaler, aims to keep the VMs operating at 100% of its capacity.

Oscillation: A combination of both undesirable effects. Oscillation occurs when scaling actions are carried out too quickly, before being able to see the impact on each scaling action on the application. Keeping a capacity buffer (aiming to keep the VMs at, say, 80% instead of 100%), or using a cooldown period (e.g. [48]) are common approaches to avoid oscillation.

The auto-scaling process matches the MAPE loop of autonomous systems [49] [50], which consists of four phases: Monitoring (M), Analysis (A), Planning (P) and Execution (E). First, a monitoring system gathers information about the system and application state. The auto-scaler encompasses the analysis and planning phases: it uses the retrieved information to make estimations of future resource utilization and needs (analysis), and then plans a suitable resource modification action, e.g., removing a VM or adding some extra memory. The provider will then execute the actions as requested by the auto-scaler. Each of the phases in the MAPE loop is described in more detail in the forthcoming sections.

Monitor

An auto-scaling system requires the support of a **monitoring system** providing measurements about user demands, system (application) status and compliance with the expected SLA. Cloud infrastructures provide, through an API, access to information useful for the provider itself (for example, to check the correct functioning of the infrastructure and the level of utilization) and the client (for example, to check the compliance with the SLA).

Auto-scaling decisions rely on having useful, updated *performance metrics*. The performance of the auto-scaler will depend on the quality of the available metrics, the sampling granularity, and the overheads (and costs) of obtaining the metrics [51]. In the literature, authors have used a variety of metrics as drivers for scaling decisions. An extensive list of those metrics is provided in [52], for both transactional (e.g. e-commerce web sites) and batch workloads (e.g. video transcoding or text mining). They could be easily adapted to other types of applications.

Hardware: CPU utilization per VM, disk access, network interface access, memory usage.

General OS Process: CPU-time per process, page faults, real memory (resident set).

Load balancer: size of request queue length, session rate, number of current sessions, transmitted bytes, number of denied requests, number of errors.

Application server: total thread count, active thread count, used memory, session count, processed requests, pending requests, dropped requests, response time.

Database: number of active threads, number of transactions in a particular state (write, commit, roll-back).

Message queue: average number of jobs in the queue, job's queuing time.

Note that, when working with applications deployed in the cloud, some metrics are obtained from the cloud provider (those related to the acquired resources and the hypervisor managing the VMs), others from the host operating system on which the application is implemented, and yet others from the application itself. In order to reduce the complexity of the monitoring system, sometimes *proxy* metrics are used: for example, CPU utilization (hypervisor-level) as a proxy of current system workload (application-level).

As stated before, auto-scaling is mostly related to the Analyze and Plan parts of the MAPE loop. For this reason, monitoring will not be further studied in this review. It will be assumed that a good monitoring tool is available, gathering different and updated metrics about system and application current state, with negligible intrusion, and at a suitable granularity (e.g. per second, per minute).

Analyze

The **analysis** phase consists of processing the metrics gathered directly from the monitoring system, obtaining from them data about current system utilization, and *optionally* predictions of future needs. Some auto-scalers do not perform any kind of prediction, just respond to current system status: they are *reactive*. However, other use sophisticated techniques to predict future demands in order to arrange resource provisioning with enough anticipation: they are *proactive*. Anticipation is important because there is always a delay from the time when an auto-scaling action is executed (for example, adding a server) until it is effective (for example, it takes several minutes to assign a physical server to deploy a VM, move the VM image to it, boot the operating system and application, and have the server fully operational [53]). Reactive systems might not be able to scale in case of sudden traffic bursts (e.g. special offers or the *Slashdot effect*). Therefore, proactivity might be required in order to deal with fluctuating demands and being able to scale in advance.

Part of the reviewed works focus only on this analysis phase, on the way of processing the metrics to either determine the current state of the application or anticipate future needs.

Plan

Once the current (or future) state is known (or predicted), the auto-scaler is in charge of **planning** how to scale the resources assigned to the application in order to find a satisfactory trade-off between cost and SLA compliance. Examples of scaling decisions are removing a VM or adding more memory to a particular VM. Decisions will be made considering the data obtained from the analysis phase (or directly from the monitoring system) and the target SLA, as well as other factors

related to the cloud infrastructure, including pricing models and VM boot-up time.

Execute

This phase consists of actually **executing** the scaling actions decided in the previous step. Conceptually, this is a straightforward phase, implemented through the cloud provider's API. Actual complexities are hidden to the client. Remember that it takes some time from the moment a resource is requested until it is actually available, and that bounds on these delays may be part of the resource SLA.

2.4 Noisy-neighbor effect and Performance Anomaly Detection

Cloud providers try to maximize the utilization of their data center. This usually means packing (or even over-packing) applications in a few physical servers. Application users rarely use 100% of the resources that they are paying for, and thus, cloud providers can make some extra money from it. However, there will be times when applications do need all the allocated resources and then, conflicts appear to access resources. Virtual Machines compete for the same resource, causing interferences among them. This problem is called *the noisy neighbor problem*. It creates performance issues in the affected applications and its VMs/containers.

2.4.1 Noisy Neighbor Problem

Cloud data centers are able to run millions of applications [7]. Each application service or task is typically encapsulated in a Virtual Machine (VM) or container. VMs or containers with different resource usage needs may be co-located in the same physical machine. Resource sharing (including CPU, memory or cache) may cause resource contention bottleneck, i.e., two VMs (or group of containers) compete for the same resources, but the resource capacity is not enough for both of them. This leads to *anomalies* in the resource usage of the application that may penalize application performance.

Resource management in virtualized environments typically makes use of consolidation or overbooking techniques, which just increments the risk of VM interference. Application malfunctioning translates directly into financial penalties: end-users will be discouraged from using the application, or cloud providers must compensate the client for SLA (Service Level Agreement) violations. These are some real numbers from large corporations [54]: Amazon suffers from 1% decrease

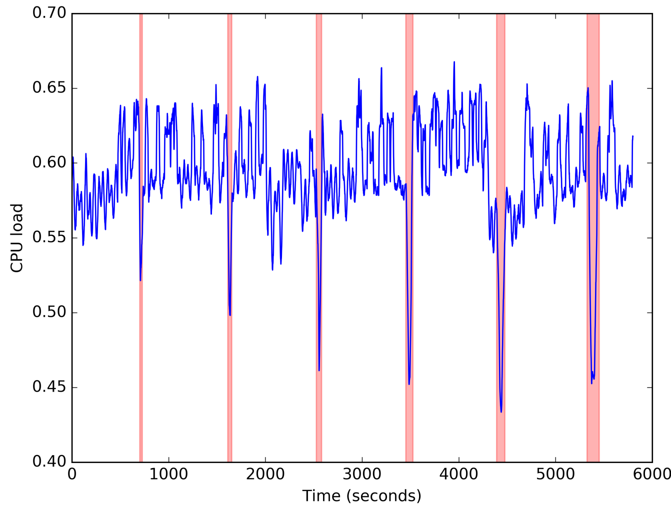


Figure 2.1: Sample noisy neighbor effect on CPU load. Anomalies are shaded areas (in red color). Data has been collected from a benchmark application (a static website) running on a Docker container. Anomalies represents interference caused by another process or container; the benchmark application cannot access to the CPU share that was initially allocated to it.

in sales for additional 100 ms delay in response time, while Google reports a 20% drop in traffic due to 500 ms delay in response time. Thus, detecting performance issues is mandatory from the cloud provider's perspective, in order to avoid performance degradation that might cause significant economical losses.

VMs/containers in the same physical server share some resources, such as CPU, memory, or cache hierarchies. Resource sharing may lead to VMs/containers affecting or being affected by other co-located VMs/containers. The *noisy neighbors* effect is an analogy for this interference [55]. It is reflected as anomalous resource usage from the service running inside the VM or container (see Figure 2.1). Its detection imposes several challenges: (1) the normal resource usage pattern is application-specific, with unknown distribution, and (2) prone to change due to workload variations; and (3) even the *anomaly* definition is application-specific and it might appear under different forms.

2.4.2 General strategies to address it

There are different alternatives to address the noisy neighbor problem. Some lie under the mitigation or avoidance approaches [56, 57, 58], others in the detection

side [26, 59?].

The first obvious technique is to try avoid the problem. Several solutions have been proposed to deal with performance issues coming caused by resource interference. There exist techniques to provide particular resource isolation (mitigation), i.e. CPU pinning [60]. Schedulers might try to select applications with compatible profiles(e.g. CPU-intensive with memory-intensive one). Still, cloud data centers are highly dynamic due to different factors e.g. rapid elasticity [61], VM migrations, varying incoming workloads. Thus, anomalies in performance will happen, and detection algorithms are a *real* need in data centers.

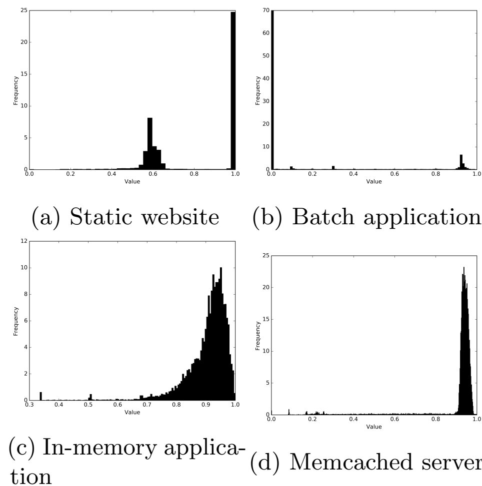
There are two main strategies in any detection problem, supervised and unsupervised approaches. Supervised learning algorithms are suited for recognizing well-known anomalies, but they require labeled datasets. It is difficult, if not totally impossible, to obtain labeled training data (i.e., measurement samples associated with normal or abnormal labels) from production cloud systems [62]. The need for well-labeled training data greatly limit the scope of their application for real-time use [63]. In contrast, unsupervised learning algorithms require no labeled data. In addition to this, such algorithms are particularly suitable for detecting unknown anomalies in cloud data centers where precise definition of anomaly characteristics may not always exist [63].

Anomaly detection methods can be classified into two subcategories, which are threshold-based and statistical methods [?]. Threshold-based approaches are too simple and usually ineffective when anomaly properties vary over time. In contrast, statistical methods are often used for anomaly detection in the cloud [62, 64?], but they usually suffer from high computing overheads and often require prior knowledge about application [?].

CPI2 [65] approach falls under this category of statistical detection methods. Authors propose adjusting a distribution probability to the CPI metric (generalized extreme vale in their case), and take values that exceed 2σ (or 3σ) from the mean as outliers. However, we can easily see in Figure 2.2 that resource usage (CPU in this case) of different applications may show different probability distributions. Therefore, an anomaly detection algorithm cannot just assume a concrete probability distribution.

2.4.3 State-of-art

This section focuses on the state-of-art and contrasts it against our proposal. First, we focus on the different approaches to tackle the noisy neighbor effect, mainly by prevention or posterior detection. Second, we analyze different detection techniques for anomaly detection in the cloud (threshold-based and statistical approaches). Given that our proposed solution falls under the category of statistical techniques, second subsection is devoted to analyze the state-of-start of these type



(a) Static website (b) Batch application

(c) In-memory application (d) Memcached server

Figure 2.2: Histogram of CPU load for different applications

of techniques to solve the Noisy Neighbor effect. Finally, we make a summary of the pros and cons of our proposed algorithm.

Interference problem or Noisy neighbor effect

In cloud environments several VMs/containers can be co-located within a server which may lead to interference among them, and consequently to disturbances in their performance due to their competition for accessing shared resources, such as cache or memory. Common techniques in data center management, such as resource overbooking or VM consolidation [66], even increase the risk of performance interference. In fact, the work presented in [67] concluded that queuing delays, scheduling delays and load imbalances all significantly impact performance. Therefore it is of great importance to detect and actuate as soon as the performance anomaly is detected.

There are works focusing on detection and mitigation of these interference, such as [56], [57], or [58], however they are usually threshold-based and are mostly focused on detecting resource shortages rather than interference itself or deviations from normal behavior. Another interesting work is presented in [65], where the normal behavior of applications is characterized based on their CPI overtime, and then it detects (and tries to alleviate) if applications deviate from their expected CPIs, but it requires previous information about CPI models for each specific application.

In addition to that, there are also works targeting the avoidance of these

anomalous situations, such as the approach presented by Delimitrou et al. [68] or by Tomás et al. [69] where different scheduler are proposed to classify and allocate the incoming applications based on their profiled and expected interference with other already running VMs. Nevertheless, some interference are hard, if not impossible, to predict and usually workload dependent. Consequently, their work could be applied together with ours, as there is still need for detecting anomalies even if you are trying to avoid them, since they may still happen.

Techniques for anomaly detection

Many techniques has been applied for the anomaly detection problem, and more specifically, to the noisy neighbor effect. These techniques can be classified into different categories following different criteria. Here we will explore two classifications: supervised and unsupervised, depending on the availability of tagged datasets, and threshold-based vs statistical techniques.

Supervised vs unsupervised approaches The anomaly detection problem can be addressed with two general approaches, supervised and unsupervised one. Supervised approach can be performed when anomalies are well-known, that is, there are available datasets with labeled anomalies. Examples of supervised methods include random forests or support vector machines [59]. Bodik et al. [26] rely on historical performance data collected during a performance crisis, and used it to create a fingerprint, that is, a signature of the anomalous performance behavior. This approach is hard to apply when there are no previously known crises. Actually, as Dean et al. [62] state, it is difficult, if not totally impossible, to obtain labeled training data (i.e., measurement samples associated with normal or abnormal labels) from production cloud systems.

Unsupervised algorithms seem particularly suitable for detecting unknown anomalies in cloud data centers where precise definition of anomaly characteristics may not always exist [63]. Thus, we selected an unsupervised approach for our anomaly detection algorithm. This approach has also been used in the literature [62, 64]. The closest article to our proposed detection method is [64], in which they develop a clustering-based technique for anomaly detection. However, we prefer to apply clustering to all resources at once, in order to capture anomalies that affect several resources simultaneously. Other similar approach is UBL [62] that relies on the use of Self-Organizing maps to capture the normal behavior of the VM. Still, they required data labeled as normal to train the model. Besides, they need some pre-training (cross-validation) to find the correct initialization.

Threshold-based vs statistical techniques Initially, we determined a set of four requirements for any anomaly detection solution, that are not usually met by

state-of-art methods. Those methods are: (1) application-agnostic, (2) suitable not unknown metric distributions, (3) online and lightweight, (4) accounting for metric correlations.

Anomaly detection methods can be classified into two subcategories, which are threshold-based and statistical methods [**?**]. Threshold-based approaches consist of directly setting a threshold over a metric. This is usually ineffective, as application resource usage pattern changes overtime and even anomaly properties do not remain the same.

Statistical techniques are by far the most prominently method used to detect performance anomalies and identify associated bottlenecks [63], and they are usually applied in the cloud. However, statistical approaches suffer from high computing overheads and often require prior knowledge about application [63**?**]. Few of these methods can deal with the scale of future cloud systems and/or the need for online detection [**?**].

Some offline approaches have been proposed. Liu et al. [70] perform an offline analysis of historic data to detect bottlenecks. Although this kind of analysis might be useful to determine the characteristics of previous anomalies happened in the system, an offline approach does not fulfill one of our requirements. Anomaly detection methods must be detected as soon as possible, and thus, the algorithm needs to be lightweight and require minimal training data.

A common statistical approach to anomaly detection consists on modeling the normal behavior of the application, and based on it, determine a threshold for anomalies. Wood et al. [58] use a probability distribution (generated based on histogram) and a time series to capture the resource usage profile of each physical server. They consider each metric separately: CPU utilization, network bandwidth utilization, and swap rate. Then, an alarm is raised when e.g. the aggregated CPU utilization on the physical server exceed a threshold. However, setting this threshold is highly dependent on each application. Alternatively, Zhang et al. [65] propose adjusting a distribution probability to the CPI metric (generalized extreme vale in their case), and take values that exceed 2σ (or 3σ) from the mean as outliers. However, we have already seen that the resource usage profile is application-specific (see Figure 2.2).

Previous approaches Zhang et al. [65] Wood et al. [58] cannot be easily generalized to all any application, in contrast to our algorithm. Besides, they do not model correlations among metrics. Gaussian Mixture Models seem fulfill these two requirements. Singh et al. [71] use GMM, because they can be used to fairly represent different types of load distributions as a convex combination of several normal distributions with respective means and covariances (thus, also taking into account for metric correlations). Similarly to our approach, Yu et al. [72] leverage GMM to represent the normal machine performance and then, calculate a distance between the GMM for the most recent observed machine condition and

that for normal machine operation. Here we can identify two main issues. First, the normal behavior of an applications evolves constantly due to workload changes, and thus, our approach of using a sliding training window is more effective. The second issue would be to determine a anomaly threshold for the distance between the current state and the normal state. Instead, we effectively avoid this threshold by comparing distances at short and long term lags.

Chapter 3

Modeling the Cloud Data Center

We already said that cloud computing is a business model for a virtualized data center. A more formal definition is provided by NIST and states that *Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.* [31]

Cloud data centers are composed of thousands of physical servers connected via a complex network topology. Before we deal with the problem of how to choose the best placement for cloud applications in the physical servers, we will describe the Data Center environment and proposed some model that can be used for later testing.

Three main roles can be identified in a cloud computer environment: the provider, the client and the end-user. The *provider* is the owner of the infrastructure, the set of data centers that host resources and leases them to its *clients*. These in turn are the ones paying for the utilization of the collection of VMs (and other resources) on which their applications run. These applications serve the requests submitted by many *end-users*. For the purpose of this paper, this is the way the three parties interact:

1. A client wants to allocate an application in the cloud. For this purpose, he will lease some resources (VMs) from an IaaS provider. The client will select a set of VM types with different capacities, including (among others) the network bandwidth. This interaction is carried out using a management API.

2. Given the set of VM types, and (if provided) the communication pattern of

the particular application, the provider will decide how to allocate the VMs onto the different physical servers in the data center.

3. Once the application has been allocated and deployed on the assigned resources of the data center, it is ready to serve requests from end-users. For example, in case of a web application, end-users will send HTTP requests generated by their browsers.

4. Each end-user request arriving to an application will trigger the execution of pieces of code in one or more VMs, as well as some inter-VM communication (using part of the available bandwidth) for example to carry out a database query.

From the point of view of the provider, the interaction with the clients consists of requests to acquire/release collections of VMs, which impact on the allocation of resources done by the provider. In contrast, from the point of view of the application, the interaction with the end-users is by means of HTTP requests/responses, which act on the use of the resources on which the application is running.

This section presents several models used throughout this work to characterize and solve the VM placement problem. We need (and define) models to:

1. Describe the structure of cloud-hosted applications as a collection of VMs, and a communication pattern among them (that is, an application model).

2. Describe the arrival of end-user requests to a particular application, and the way each request triggers the utilization of the resources assigned to the application.

3. Describe the structure of the data center managed by the provider, which is capable of hosting multiple applications of different clients.

4. Describe the way the use of resources translates onto the use of power (that is, a power model), considering as resources CPUs (physical servers) and network equipment (switches).

3.1 Modeling Applications

A client willing to run an application in the cloud will interact with the provider, requesting a set of VMs. We assume that the application is built using a layered organization, with communication between layers. Also, we assume that the client knows the structure of his application, and that this structure is part of the

resource request sent to the provider. The provider may take into account the application structure in order to make an optimal allocation.

In our model, the definition of an application requires several parameters: the number of layers (L), the number of VMs in each layer (N_i being i the layer identifier) and a matrix of the communication needs (or bandwidth, measured in Mb/s) between each pair of VMs i and j ($BW = [bw_{i,j}]$), and with the external world.

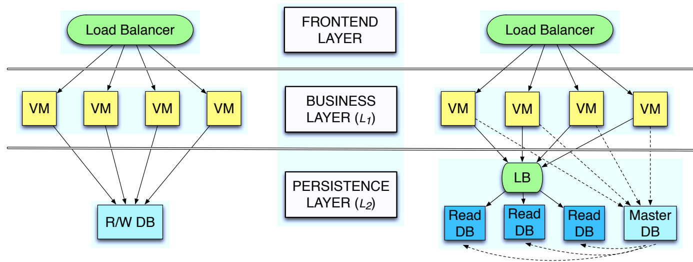


Figure 3.1: Two web application types: L-WA, with a single R/W database server (left); H-WA, with a master-slave database system (right).

For this work, we particularize this model to define two classes of web applications (see Figure 3.1). Web applications are usually implemented using a three-layer architecture:

1. A load balancer that receives end-user requests and distributes them evenly along the VMs of the business layer; it may be implemented on a hardware device, or as a DNS-based redirection—thus, we do not include it in the application model.

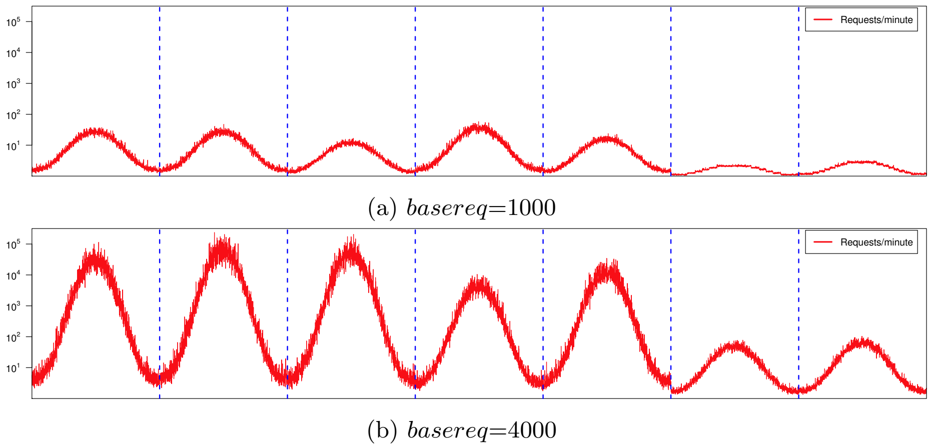
2. A business layer that contains the logic of the application. It comprises a pool of VMs that processes the input requests (and generates the corresponding replies) redirected by the load balancer. The number of VMs at the business layer depends on the intensity of the workload generated by end-user requests.

3. A persistence layer that processes the database (DB) requirements of the application. The number of VMs in the persistence layer depends on the intensity of the workload generated by the application. A light workload can be managed by a single DB server that supports both read r and write w operations; we represent this class of applications as *L-WA*. For applications with heavy database demands (*H-WA*), a master-slave replication scheme

may be applied: one of the VMs of the persistence layer is the *master* node that processes all the write w operations, while the read queries r are evenly distributed along the remaining VMs in the layer. Whenever a change (write w) is made on the master node, it is propagated to the remaining DB replicas.

IaaS providers typically offer different predefined types of VMs, with different resource sets, called *instances*. In this work, we will consider *small*, *medium* and *large* instances, with different characteristics only in terms of allotted network bandwidth (in Mb/s): bw_s= 50, bw_m=150 and bw_l=300, respectively. For *L-WA* web applications, we consider that the business layer (L_1) uses small instances and the database layer (L_2) contains a single, large VM. For *H-WA* applications, the database is modeled as a single large instance for the master DB node, and several medium-size VMs for read DB nodes.

3.2 Modeling Application Workloads



(a) *basereq*=1000

(b) *basereq*=4000

Figure 3.2: Two examples of HTTP workloads generated by end-users, with different values for *basereq*

Once the application has been placed in the data center, it is ready to serve requests coming from end-users. Our model considers that a request arriving to a web application can be of one of these three types: (1) p: it is processed in the business layer, and requires no access to the database; (2) r: it requires a

query (read operation) to the database; (3) w: it requires a write operation on the database.

Each request type implies different inter-VM messages and processing times in the different layers of the application. For example, an r request requires the following execution steps:

1. The initial HTTP request sent by the end-user is redirected (by the load balancer) to one of the VMs of the business layer (d_0)

2. After an initial processing time in the business layer, a query is sent to the database layer (d_1)

3. The database server will process the request (again, using some CPU time) and will send the response back to the business layer (d_2)

4. Finally, the VM in the business layer sends the response to the user, with the HTML content (d_3)

Each of the messages involved has an associated size (d_0, d_1, d_2, d_3) that is generated randomly using the ranges defined in Table 3.1. These values have been extracted from a report [73] that analyzes billions of web pages. The transfer rate will depend on the bandwidth allocated to each particular VM.

Table 3.1: Data size ranges (in KB) of each message, for each request type

Req type	d_0	d_1	d_2	d_3
p	5-10	0	0	5-500
r	5-10	10-50	20-400	5-500
w	5-10	10-50	20-300	5-500

The response time of web applications is usually in the order of milliseconds. For our simulation-based experiments, the CPU processing time of each request is generated randomly, in the range of 50-150 ms per tier. p-type requests only require 50-150 ms in the business tier, while both r and w requests imply: 50-150 ms in the business tier, another 50-150 ms to execute the database read/write query, and 50-150 additional ms in the business tier to finally send the response back to the end-user. These values have been extracted from [74] [75]. Note that given the time-sharing nature of the CPU resource, several requests may be executed concurrently on the same VM and, thus, the processing time will increase with the degree of concurrency.

For this work we propose a workload generation function designed to generate sequences of HTTP requests, that will be used in our simulation-based experiments

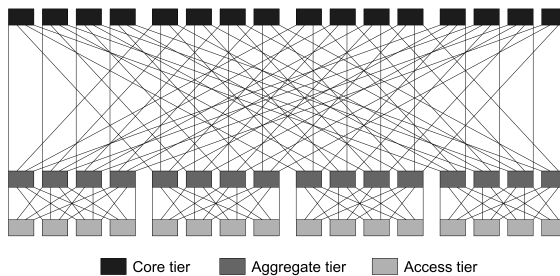
to "feed" applications deployed in the data center. The model is described in Algorithm 1. The generated application workload follows a diurnal pattern, with a lower average request rate at weekends (see Figure 3.2 for some examples). An initial per-day *basereq* parameter (baseline number of requests) is customized for each application, which is lowered for weekend days. Then, a sinusoidal function [76] [77] is used to generate a base number of requests per second. Using different probability distributions and ranges, we add burstiness to the workload. Finally, a request type (p, r or w) is selected randomly, together with the size of the request. In Figure 3.2, we have plotted two sample application workloads that follow a daily pattern.

```
Data: Timestamp, basereq
Result: List of reqs (reqType, data)
reqs = 0;
r = 0;
if dayOfWeek(timestamp) is Monday-Friday then
|  basereq = basereq + rnd(-basereq*0.30, basereq*0.30);
else
|  basereq = rnd(basereq*0.30, basereq*0.50);
end
reqs = (basereq/2) * sin((2πtimestamp)/PERIOD + 4.75) + (basereq/2);
r = rand(0,1);
reqs += createBurstiness(r);
reqList = list();
foreach req do
|  reqType = getReqType(appType);
|  reqData = getDataSize(appType, reqType);
|  add(reqList, newReq);
end
```

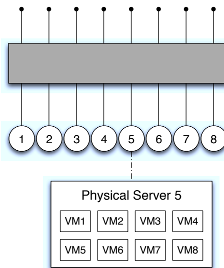
Algorithm 1: Per-application process to generate end-user HTTP requests

3.3 Describing the Data Center Structure

As previously stated, current data centers are usually built using tree-based topologies, such as fat tree and VL-2 [41]. This kind of networks are composed of several tiers of switches (we assume homogeneous switches) and several servers connected to the bottom tier of the tree (the edge or *access* tier). Each server is divided into several *slots*, where each slot can be a fraction of a core, an entire core or several cores. Application VMs are assigned to different slots of the data



(a) 3-tier fat tree built with 8-port switches



(b) 16-port access switch

Figure 3.3: Representation of the physical configuration of a data center (network and servers).

center servers. Throughout this work we assume that a VM consumes a slot, and that one slot is equivalent to one core of a multi-core server.

The physical configuration of a data center is defined as the number of servers (P), the number of cores (slots) per server (C_p) and the network topology. In particular, a tree-based topology is defined by the number of uplinks and downlinks of the switches (S_{up} and S_{down}), the bandwidth (Mb/s) offered by each switch port (S_{bw}) and the number of tiers of the tree (T). The communication latency between two cores i and j depends on the distance between them, measured in terms of *hops*. Given a topology, the distance $d_{i,j}$ between a any pair of cores (actually, any pair of servers) can be computed. A per-data center matrix $D = [d_{i,j}]$ summarizes all these distances.

In this work we have focused on interconnection networks built using fat trees. The main characteristics of these trees are the low average path length and the availability of multiple paths between nodes, thus performing well with almost any king of workload. Therefore, it is the network of choice in high-performance data centers. The particular fat tree network that we have used is composed of three

tiers (as depicted in Figure 3.3a) with the same number of switches in each one of them (see Figure 3.3b). We consider that core switches are directly connected to the Internet,the bandwidth of this connection being the aggregated bandwidth of all the switches. Also, we assume that the network interfaces of the servers in the access tier are of capacity S_{bw}. In this kind of tree the distance between two cores/servers is computed as follows: cores in the same physical server are at distance 0; servers connected to the same access switch are at distance 2; if aggregation or core switches are required, the distance grows to 4 or 6 respectively.

The configuration parameters used in the experiments to model the data center are those specified in Table 3.2.

Table 3.2: Parameter configuration for the data center infrastructure

Par	Value	Description
P	512	Number of servers
C_p	8	Cores (slots) per sever
T	3	Number of tiers in the fat-tree
S_{up}	8	Number of uplink ports per switch
S_{down}	8	Number of downlink ports per switch
S_{bw}	1000	Capacity of links (Mb/s)

3.4 Modeling Power Requirements

Nowadays, the reduction of the power consumption of data centers is receiving increasing attention. Energy is consumed by servers and switches, and also by cooling and energy distribution systems. Reducing power use has direct benefits for the infrastructure provider (lowering the energy bill), while reducing the carbon footprint of the data center.

PowerNap [78] aims to reduce the consumption of unused servers by switching off memory, disk and other elements. In this work we assume that a strategy like this is used in the data center: unloaded servers and switches operate in an idle state that minimizes energy waste. We define a general model of power utilization of a device (server or switch), inspired on that provided in [78].

$$E = \begin{cases} E_{idle} & U_{active} = 0 \\ E_{active} + \frac{E_{rem} \cdot (U_{active} - 1)}{U_{total} - 1} & U_{active} > 0 \end{cases}$$

Table 3.3: Parameter values of energy utilization in physical servers and switches.

Symbol	Consumption at	Server (W)	Switch (W)
E_{max}	100% utilization	200	100
E_{idle}	Idle state	10	10
E_{active}	One active core/port	160	31
E_{rem}	Remaining $U_{active} - 1$ cores/ports	40	69

The energy consumption E of a server/switch (in Watts) depends on the number of active cores or ports U_{active}. In an idle state, the consumption is equal to E_{idle}. The transition from the idle state to the activation of the first core/port implies an important increase in the energy utilization, because it requires turning on other resources (memory, disk) or internal fans. The activation of additional cores/ports causes a linear increase of used power. This energy model is similar to the one proposed by Reviriego et al. [79]. In this work we do not consider the penalty time due to the transition from the idle to the active state. As the authors state in [78], this time is negligible for jobs that last more than 10 ms using this model, which is actually our case.

Table 3.3 shows the energy consumption values used in the experiments, both for servers and switches. Values for servers are based on those in [78]. Values for switches are taken from manufacturer data sheets, and are similarly to those reported in [40].

Chapter 4

Application Placement: An Optimization approach

This is a collaborative work. The contributions of the author lie on the problem definition, formulation and modeling, and also in the definition of the problem-specific operators, specially the guided crossover.

In recent years, the utilization of cloud infrastructures to host applications has widely spread. The characteristic that makes these cloud systems so appealing is their elasticity, that is, resources can be acquired on demand, depending on the needs of the time-varying application, but paying only for those actually booked (a scheme known as pay-as-you-go). Virtualization technologies enable the cloud infrastructure to provide such elastic usage. The resources offered by physical servers, organized in several data centers, are provided in the form of abstract compute units that are implemented as Virtual Machines (VMs). Each VM is assigned a pre-configured set of resources, including: number of cores, amount of memory, disk and network bandwidth.

Virtualized data centers support a large variety of applications, including batch jobs (typically used for scientific applications), and web applications (e.g. an online bookshop). Each application is deployed on a set of VMs, which can be allocated to any collection of physical servers in the data center. The problem of assigning a physical location to each VM is known as *VM placement* and it is performed by the manager of the cloud infrastructure. This manager is typically called the Infrastructure-as-a-Service (IaaS) provider.

The challenge for the provider is to host a large and diverse set of applications (VM sets from different clients) in the infrastructure trying to (1) maximize its revenue and (2) provide a good service to the clients. An adequate application

placement would be able to maximize the resource usage of physical servers and reduce the energy consumption of the data center, for example, by turning off (or setting to idle state) the inactive servers and network elements (typically, switches). At the same time, the infrastructure management policies should trade off the obtained revenue with the Quality of Service (QoS) agreed with the client, guaranteeing that each application receives the resources paid for.

The VM placement problem has been extensively explored in the literature (e.g. [38] [39] [41] [42]). Most efforts have been directed towards optimizing the usage of CPU, memory and disk resources, and reducing the energy consumption of physical servers. However, not enough attention has been paid to the utilization of the network. An inappropriate placement of VMs with heavy communication requirements could lead to the saturation of certain network links, with the subsequent negative impact on applications (longer execution or response times). Besides, as stated in [40], the network power has been estimated at 10-20% of the overall power consumption. For this reason, the VM placement policy should try to reduce not only the use of physical servers, but also the use of network links and switches to reduce the total power footprint.

The most common topology of data center networks is a tree of switches arranged in several tiers. The communication latency of any pair of VMs depends on the distance between the physical servers in which they are allocated. This, in turn, depends on their position in the tree. Distance is measured as the number of hops from the sender VM to the recipient. The collection of VMs forming an application communicate among them following a certain communication pattern. In web applications, the VMs are arranged into several layers and there may be intra- and inter-layer communication. Other patterns are possible, depending on the particular characteristics of the application.

Based on the communication pattern of an application, and with an estimation of the workload imposed by its end users, it is possible to approximate the input/output network bandwidth needed by each VM. The most communicative VM subsets should be placed as *close* as possible (minimizing the distance between them in terms of network hops). This means using the minimum number of physical servers, because intra-server communication is the cheapest. The constraint is that the external aggregated bandwidth required by all the VMs in a server, from the same or from different applications, cannot exceed the bandwidth of its network connection.

Two examples of common VM placement policies that are used in data centers are first fit (FF) and round robin (RR). Each of them has a different characteristic, that the infrastructure manager has to consider to choose the “right” one. FF simply selects the required physical resources consecutively, starting from the first available server. The use of this policy results in a higher utilization of the active servers because it tends to fill the possible gaps inside them. For the same reason,

the number of active servers is expected to be reduced, thus saving energy. RR tries to equalize the utilization of all servers to avoid excessive wearing-out of server subsets and thermal peaks. The chosen policy affects not only the use of the infrastructure, but also the applications running on it. In the long term, FF tends to place applications in several, non consecutive nodes ("consecutive" has to be read in terms of network position), whereas RR favors contiguous placement of all the VMs of an application. This has an impact in the use of network elements (and the network-related power used) and on the performance of the applications.

In this work we demonstrate how it is possible to take these policies as starting points and use evolutionary optimization techniques to find placements that improve the benefits for both the infrastructure provider and the application. In particular, we evaluate three well-known multi-objective evolutionary algorithms with problem-specific crossover and mutation operators. They implement mechanisms to converge rapidly to high quality placements. Experiments demonstrate that allocating applications using optimization-based policies results in a lower utilization of resources (servers, networking elements) while improving the performance of applications.

4.1 Multi-objective Optimization Algorithms

4.1.1 Why multi-objective approach?

Let us assume that the problem consists of n objective functions that we want to minimize (the opposite, maximizing is computed similarly)

How do we combine different objectives into one? Easy approach:

$a_1 f_1 + a_2 f2 + \ldots + a_n f_n$

But how do we determine the coefficients $a_1, a_2, \ldots, a_n$?

Instead, the multi-objective approach determines the set of non-dominated solutions, called Pareto set or Pareto front. Non-dominated means that for this particular solution, none of the objectives functions can be improved without penalizing others.

4.1.2 Algorithms

This section describes the optimization algorithms used in this work to solve the problem of the initial VM placement, formulated as a multi-objective problem. We have selected three multi-objective evolutionary algorithms: NSGA-II, SPEA2 and Hype.

At each step of the optimization process (called *generation*), each of the algorithms maintains a set (*population*) of individuals (candidate solutions for a given VM placement problem). The quality of a solution is assessed using several

fitness functions (objectives) that represent the (possibly constrained) problem; in this case, the placement of a collection of VMs. At each generation, the most promising individuals are chosen using a selection criterion and included in the new population. The offspring is generated by applying crossover and mutation operators over the individuals of the current population. The optimization process is repeated until the stopping criterion is fulfilled.

The result of this optimization process is a set of solutions that simultaneously optimize each of the objectives (called Pareto set). The value of the functions achieved by the Pareto optimal solutions is called Pareto front. Formally, we define a multi-objective optimization (minimization) problem subject to some restrictions as:

$$min\{f_1(x), \ldots, f_{N_{Obj}}(x)\} \tag{4.1}$$

$$\begin{cases} g_j(x) = 0 \; j = 1, \ldots, M, \\ h_j(x) \leq 0 \; j = 1, \ldots, K \end{cases} \tag{4.2}$$

where f_i is the $i - th$ objective function, x is a vector that represents a solution, N_{obj} is the number of objectives, $M + K$ is the number of constraints, and g_j and h_j are the constraints of the problem.

Now we explain each of the three optimization algorithms tested in this work. The main difference between them relies on the selection criterion used to choose the best candidate solutions at each generation.

- Non-dominated Sorting Genetic Algorithm II: The aim of the NSGA-II [80] algorithm is to improve the adaptive fit of a population of candidate solutions to a Pareto front constrained by a set of objective functions. The population is sorted into a hierarchy of sub-populations based on the ordering of Pareto dominance. Similarity between members of each sub-group is evaluated on the Pareto front, and the resulting groups and similarity measures are used to promote a diverse front of non-dominated solutions.

- Strength Pareto Evolutionary Algorithm 2: SPEA2 [81] uses as selection criterion a combination of the degree to which a candidate solution is dominated (strength), and an estimation of density of the Pareto front as an assigned fitness. An archive of the non-dominated set is maintained separate from the population of candidate solutions used in the evolutionary process, providing a form of elitism.

- Hypervolume Estimation Algorithm: Hype selects the best candidate solutions using the hypervolume indicator. This measure is consistent with the concept of Pareto-dominance; the set of solutions with the highest value

of the indicator dominates other sets. However, the calculation of the hypervolume requires a high computational effort. Hype [82] addresses this issue trading off the accuracy of the measured estimates with the available computing resources.

4.2 Topology-aware Optimization

The aim of this work is to find a suitable placement for the VMs forming an application onto a set of available cores (slots) in the data center servers. Taking as starting point a VM placement obtained using a simple policy, either FF or RR, we will use a bi-objective optimization algorithm to obtain a better one. This optimization takes into account the communication needs of the application being allocated, as well as the structure of the data center, aiming to benefit the cloud client (by making its application perform better) while allowing the cloud provider to save energy costs. In this section we focus on the formal definition of this particular optimization problem, as well as on the specific crossover and mutation operators needed by the three multi-objective optimization algorithms being evaluated.

4.2.1 Problem Definition

Given an application A with a VM set V of size N, and a subset of available cores $C' \subset C$, where C is the whole set of cores in the data center (note that usually $|C'| >> N$), the VM placement problem involves finding a mapping function φ that assigns each VM, $v \in V$ to a core $c \in C'$:

$$\begin{aligned} \varphi :& V \rightarrow C' \\ & v \mapsto \varphi(v) = c \end{aligned}$$

A solution to the VM placement problem has the form $s = (s(1), \ldots, s(N)) = (c_1, c_2, \ldots, c_N)$ representing that the VM i is assigned to core $s(i) = c_i$.

Two major selection criteria will be considered to choose a VM placement. First, we favor solutions that minimize communication latency. For this reason, the VM placement will try to allocate the most communicative VMs onto close cores, in terms of network distance. The second criterion focuses on reducing the number of servers allocated to the application. An allocation solution that fulfills the first criterion may not satisfy the second one. For example, given an application $A = \{v_1, v_2, v_3, v_4\}$ in which communication occurs between v_1-v_2 and v_3-v_4, the first criterion may place each pair of VMs on a different physical server. However, according to the second criterion, it would be better to place all the VMs

in the same server. Both criteria try to positively impact the use of data center resources, by means of reducing the number of active servers and switches, but the first one specifically tries to benefit the application, optimizing its performance. Placement solutions must obey a restriction: external communication demands of all the VMs assigned to a server cannot exceed the bandwidth of its network link S_{bw}. This constraint does not take into account communication between VMs in the same server.

More formally, we describe VM placement as a bi-objective optimization problem. The first objective function to minimize is defined as follows:

$$f_1(s) : \sum_{i,j \in V}^{N} bw_{i,j} \cdot d_{s(i),s(j)} \tag{4.3}$$

where $d_{s(i),s(j)}$ is the distance between the cores assigned to VMs i and j, and $bw_{i,j}$ is the bandwidth required by VMs i and j.

Given the function $\sigma(c) = p$ that returns the server p to which core c belongs to, and a solution s, we define the set of active servers for this solution as $P^s = \{p | \exists i \in \{1, \ldots, N\} \text{ s.t. } \sigma(s(i)) = p\}$. The second objective function to minimize is defined as:

$$f_2(s) : |P^s| \tag{4.4}$$

The requirement to guarantee that the total bandwidth usage of the VMs allocated to a server does not exceed the capacity of that server's network link can be expressed as this constraint:

$$\forall p \in P^s : S_{bw} - S_{bw}^p \geq 0 \tag{4.5}$$

where S_{bw} is the bandwidth of a server's link, and S_{bw}^p is the reserved bandwidth of server i, considering the previously allocated applications and also the new one.

4.2.2 Problem-specific Operators

As stated before, at each generation the optimization algorithms must make the current population evolve using crossover and mutation operators. In this work, we have developed specific operators that consider the characteristics of the VM placement problem.

Guided Crossover Operator

Crossover is applied with probability p_{cross}. It combines two individuals to generate a new one, taking into consideration the specific characteristics of the problem. Given two parents s_1 and s_2, the crossover operator generates a new

child *ch* as follows. We define $\phi(i,s)$ as the communication cost of VM i in a candidate solution s, considering all the destinations with which it communicates, the corresponding input/output bandwidths, and the distances:

$$\forall i \in \{1,\ldots,N\} : \phi(i,s) = \sum_{j=1, j\neq i}^{N} (bw_{i,j} + bw_{j,i}) \cdot d_{s(i),s(j)} \tag{4.6}$$

Child *ch* will be constructed taking from the parents those cores that cause the lowest communication cost. That is, for each VM i, if $\phi(i,s_1) < \phi(i,s_2)$, then core $s_1(i)$ is assigned to VM i of child *ch*. A correction step to remove any possibly repeated position (cores) of each child may be required. In that case, the repeated core will be replaced with one of the non-used cores from one of the parents.

Guided Mutation Operator

Mutation is applied with a probability p_{mut}. There are two types of mutation, and the one to apply is selected based on another probability p_{mtype}. The first type performs a simple swap between any two elements of the chosen solution, without considering cores in the same server, because this change would not affect the values of the objective functions. With probability 1-p_{mtype}, the second type of mutation is applied: one of the cores assigned to the solution is replaced with a core $c \in C'$, selected randomly from the free ones using a distance-based distribution that favors physically close cores.

Selection Criterion for a Solution in the Pareto Front

The bi-objective optimization algorithm generates a collection of solutions for a given application (Pareto set), with different trade-offs between locality and number of allocated servers. As all Pareto optimal solutions are considered equally good, a selection criterion is required to choose one. We select the solution that is most beneficial for the provider: one that minimizes the *global* number of active servers in the data center P_{active}. Note that this criterion is completely different from the one used in f_2. With f_2 we try to minimize the number of servers assigned to *a particular* application, while here we select the solution that achieves the lowest number of active servers in *the whole data center*.

4.3 Experimental Framework

This section presents the simulation-based framework used to evaluate the VM placement strategies. The experiments try to provide answers to two questions: (1) which optimization algorithm performs the best when applied to the VM

placement problem (2) what is the benefit of a network-aware multi-objective VM placement, in terms of resource usage and energy consumption.

4.3.1 Simulation Environment

The VM placement has been evaluated using an in-house developed simulator. This tool is able to simulate the dynamics of a realistic data center, using the models described in Chapter 3 for the data center topology, applications, workload generation and power consumption.

For each new application allocation request (resource acquisition request from the cloud client) arriving to the provider, the best initial placement of the collection of VMs has to be computed. A preliminary placement is generated using a simple VM placement policy: FF which searches free cores sequentially, always starting at the first one, or RR which also performs a sequential search but starting from the last core used in the previous placement. Afterwards, the preliminary placement is improved using an optimization algorithm.

The data center is initially empty: none of the resources are reserved for any application. Sequences of acquisition/release operations (new applications, applications that end) are generated in order to populate the data center. Depending on the rate of these operations, we deal with three different load scenarios: *low* (25%), *medium* (50%) and *high* (75%). These percentages indicate the average use of data center resources caused by the corresponding load.

Each experiment carried out on the simulator is divided into two phases. The first one is a warming-up phase, not used for measurements, that ends when the target load of the data center is reached and the system arrives to a steady state. The second phase, with the system in steady state, is that used for gathering metrics. It consists of ten batches with sets of 1000 operations (equally distributed between arrivals and departures). The simulator collects a variety of per-batch metrics. We have performed five repetitions of each experiment, using the same list of operations, and summarized in several tables the averaged metrics of the five repetitions.

4.3.2 Experiments to Compare Optimization Algorithms

Our first task was to identify the best multi-objective algorithm for our placement problem, from a set of three candidates: NSGA-2, SPEA-2 and Hype. To do so, we carried out a collection of 120 experiments in a *simplified* environment, i.e. the data center implements the acquisitions and releases of sets of VMs as requested by the cloud clients, but the dynamics of the deployed applications is not simulated: they do not execute end-user HTTP requests. For this evaluation we need to use two models: a description of the data center topology and the description of the structure (and communication needs) of the application. With

this set-up, we can determine which algorithm is the one providing better values for the objective functions to be optimized, and for the criterion used to choose a solution from the Pareto set. The parameter configuration for the optimization algorithms is detailed in Table 4.1. Note that for this work we have not made any particular effort to tune the parameters, and we have used the same values with all the optimization algorithms.

Table 4.1: Parameter configuration for the optimization algorithms (NSGA-2, SPEA-2 and Hype)

Parameter	Value	Description
N_{pop}	100	Number of individuals per generation
N_{gen}	100	Number of generations
p_{cross}	0.8	Probability of crossing operator
p_{mut}	0.8	Probability of mutation operator
p_{mtype}	0.5	Probability for mutation type

4.3.3 Experiments to Evaluate VM Placement in Realistic Scenarios

Once a good optimization algorithm for VM placement has been identified, we can carry out more complex experiments, which involve the interaction of all the parties of a cloud computing set-up, in a *detailed* simulation: we simulate the arrival of acquisition/release of applications from cloud clients (which trigger operations for VM allocation/release by the infrastructure manager), and we also simulate the arrival of HTTP requests from end-users to the applications deployed in the cloud infrastructure (which trigger the use of resources in the VMs running the applications and, consequently, on the servers and attached switches). In order to implement this, we need to use the data center load model (low, medium or high), a per-application model of its HTTP workload, and a per-application model of resource utilization. As in the previous set-up, we have carried out 120 simulations.

Notice that the infrastructure manager will run a placement algorithm each time an application allocation request is received, and this can be done in a straightforward way (plain FF or RR) or in combination with an optimization algorithm. In this set of experiments we will consider only the multi-objective optimization algorithm which provides the best results in the previous set of experiments.

All this simulation set-up will allow us to assess the effectiveness of placement

policies in terms of application efficiency (time to process HTTP requests), resource utilization and energy consumption. Notice that these metrics reflect more closely the actual needs of cloud users and providers.

4.4 Analysis of Results

In this section we summarize and analyze the results of the experiments reported in Sections 4.3.2 and 4.3.3.

4.4.1 Simplified Experiments: Evaluation of Optimization Algorithms

The results of the simplified experiments allow us to compare the simple placement policies, FF and RR, with the same ones enhanced with a topology-aware optimization, performed with NSGA-II, SPEA-2 and Hype. The first question to ask is whether or not optimization is effective (applications and data center provider benefit from it). Then, if the first answer is affirmative, the best optimization strategy must be selected.

Results are summarized in Table 4.2, which gathers the mean μ and standard deviation σ of the multiple simulation runs for both objective functions, f_1 (communications locality) and f_2 (number of servers assigned to the application). The number of total active servers in the data center P_{active} (P_a for short) is also included in the table.

If we focus on non-optimized policies ($non - opt$), clearly RR is better for applications, as it provides lower communication costs than FF in all scenarios (see f_1 values), while simultaneously providing better (smaller) f_2 values (number of servers per application). However, FF uses on average substantially fewer servers than RR. The most relevant result, though, is that applying optimization *always* provides better values for both objective functions, compared to the baseline, non-optimized FF and RR.

It is not easy to decide which optimization algorithm performs the best. We should consider f_1, f_2 as well as the number of active servers P_{active} used as the Pareto selection criterion. Attending to P_{active} values, non-optimized FF and RR almost always use more servers than the optimized counterparts (the single exception is NSGA-II-optimized RR for high loads). Among the optimized placements, SPEA-2 is, in a majority of cases, the one using the smallest number of servers. Furthermore, in most cases it also obtains the lowest values of objective functions f_1 and f_2.

Note that the criteria used to choose an optimization algorithm only provide *hints* about the expected benefits for applications (clients) and data center

providers. The achievable benefits, expressed in more tangible terms, are analyzed in the following section.

4.4.2 Detailed Experiments: Evaluation of the Benefits of Optimization-based Placement

The objective functions f_1 and f_2, together with the Pareto selection criterion, were designed to have a positive impact on both the applications and the data center, but we need to assess those impacts in a meaningful, measurable way. For applications, we want to know the improvements in terms of the mean execution time per HTTP request. For the data center, we are interested in achieved energy savings. For the latter we could also provide the number of active servers and switches, but energy provides a single metric that assesses how "green" our proposals are.

Figure 4.1 compares the HTTP request processing time for FF and RR without and with the optimization step carried out with SPEA-2. The figure is only for the highly-loaded data center, but values for other loads are similar because the (simulated) data center never does over-subscription: it is guaranteed that clients use the resources they pay for in an exclusive way. However, the placement policy has an impact on the inter-VM communication cost: a good placement reduces inter-VM distance and, thus, HTTP requests are performed faster. We have measured 656 vs. 802 ms for FF (optimized vs. non-optimized) and 710 vs. 793 ms for RR.

Regarding the energy consumed by the data center, we have to consider that FF-based policies (with or without optimization) always use fewer servers than the RR-based alternatives. Simultaneously, all the tested policies make *very similar* use of network links and switches, because of the fat-tree topology that distributes data movements along all upper-level switches: locality only translates into negligible gains at the access layer. This comes at no surprise because the locality function to optimize is used on a per-application basis, not taking into consideration the global use of the network. The combined result is that the placement policy using fewer servers is that consuming less power, as reflected in the measurements summarized in Table 4.3. Plain RR is more power-hungry than FF. When SPEA-2 is put to work, differences between these two strategies blur. Optimized FF is in most cases still superior, but RR is the best choice for highly loaded data centers.

4.5 Summary and Conclusions

We have demonstrated that an IaaS provider can improve the VM placement policy in use by applying an optimization strategy, achieving benefits not only

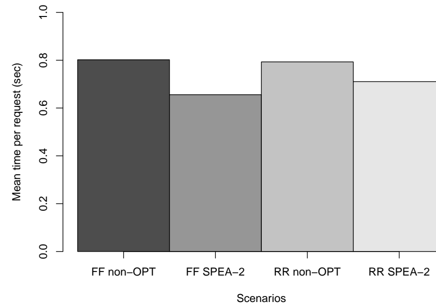


Figure 4.1: Mean execution time per HTTP request (in seconds) for the data center with high load, comparing FF and RR without and with an additional optimization with SPEA-2

for the provider but also for the client and the end-user. Furthermore, this optimization can be done at a negligible cost: it is applied when allocating a new application, taking only a few seconds. Benefits for the provider are measured in terms of used servers and switches, and immediately translate into reduced power demands (resulting in a "greener" data center). Benefits for the applications are achieved by improving their ability to process user requests. As a direct effect of improving communication latencies, the average execution time per request is reduced (up to 11–19%). Thus, the cloud client (that is, the application owner) will be better able to comply with QoS expected by end users. Simultaneously, using optimization we are able to improve the number of active servers in the data center and, therefore, the overall energy consumed: for highly loaded data centers, savings range between 7.26–13.81%. Globally, the best tested placement strategy is FF with SPEA-2 optimization, although in some instances (optimized) RR yields better results.

Table 4.2: Means and deviations of values of objective functions f_1 and f_2 and number of active servers P_a for non-optimized FF and RR (*non − opt*), and for FF and RR with an additional optimization step (using NSGA-II, SPEA-2 and Hype). Results computed for *High*, *Medium* and *Low* data center loads

		First fit					Round Robin				
		μ_{f_1}	σ_{f_1}	μ_{f_2}	σ_{f_2}	μ_{P_a}	μ_{f_1}	σ_{f_1}	μ_{f_2}	σ_{f_2}	μ_{P_a}
High	non-opt	731865.40	102999.24	37.23	3.73	354.10	702092.96	113376.77	32.33	4.25	454.53
	NSGA-II	621818.21	95383.95	35.71	2.91	341.87	699406.83	107713.29	32.15	3.36	456.46
	SPEA-2	649312.15	89880.09	35.18	3.33	339.47	660725.75	65739.05	30.04	2.72	438.10
	Hype	671846.80	115480.83	36.53	3.99	353.79	697650.12	107674.54	30.91	2.94	449.66
Medium	non-opt	722702.32	107985.19	33.53	3.44	228.65	645590.51	107245.73	21.65	2.97	303.00
	NSGA-II	607612.69	101509.97	31.52	2.34	227.98	592405.23	112321.56	19.94	2.36	283.41
	SPEA-2	630735.40	106386.83	31.69	2.54	227.57	569869.03	104114.19	20.00	2.08	286.05
	Hype	600308.65	112778.65	32.06	2.70	227.59	573890.04	120745.76	20.08	2.19	287.77
Low	non-opt	690180.40	135298.79	27.35	2.63	111.15	604539.18	114483.78	17.31	1.64	125.37
	NSGA-II	574469.29	159677.89	24.72	2.19	108.05	539529.24	145097.25	16.42	1.52	109.12
	SPEA-2	559766.55	167074.10	24.52	2.39	96.75	468476.31	115624.85	15.75	1.49	112.88
	Hype	580084.18	132498.60	25.51	2.50	104.63	509938.20	131754.41	15.88	1.58	114.21

Table 4.3: Energy consumption (in MWatts/hour) used by physical servers, switches and total, for non-optimized $Non-opt$ and SPEA-2 placements, for different data center loads

		First fit			Round Robin		
		E_{server}	E_{switch}	E_{total}	E_{server}	E_{switch}	E_{total}
High	non-opt	69.97	24.41	94.38	73.54	24.83	98.37
	SPEA-2	62.85	24.68	87.53	59.87	24.92	84.79
Medium	non-opt	45.05	25.43	70.49	70.82	25.54	96.37
	SPEA-2	42.61	25.60	68.20	49.12	25.36	74.48
Low	non-opt	43.80	26.21	70.02	68.24	26.28	94.52
	SPEA-2	42.00	26.17	68.17	45.81	26.21	72.02

Chapter 5

Taxonomy on Auto-Scaling Techniques

In this section we carry out a survey of the literature on auto-scaling systems, following the classification introduced in the previous section. It is organized in as many sub-sections as categories, each one starting with a description of the technique that defines the category, including the definition, methodologies, pros and cons. After that, we analyze a collection of papers fitting into the category, also discussing their features and limitations. This information is complemented with a set of tables summarizing the reviewed literature, one table per category.

The main contributions of this review are:

- A classification for auto-scaling techniques, together with a description of each category and its pros/cons.
- A review of the literature about auto-scaling, organized using this classification. However, given the heterogeneity of auto-scaling approaches and testing conditions, it is *not* possible to provide an assessment of the different proposals, alone or in comparative terms.

Note that the management of the cloud infrastructure is out of the scope of this paper: topics such as VM placement into physical servers, VM migration, energy consumption and related problems are not discussed.

5.1 Auto-scaling Taxonomy

As the body of literature dealing with proposals of auto-scaling systems is large, we have tried to put some order into it, to better understand and compare those proposals. To that extent, we need some classification rules to group works into meaningful sets. To the best of our knowledge, there is no previous work proposing such a classification. The most closely related survey, done by Guitart et al. [83], targets the performance of general Internet applications, deployed over shared or dedicated clusters, relying on methods such as admission control and service differentiation. However, this review focuses on exploiting the elastic nature inherent to cloud systems. Manvi and Shyam [84] gather many references about resource management on IaaS environments, but put little focus on auto-scaling.

The works we have revised to put together this survey are very diverse in regards to the underlying theory or technique used to implement the auto-scaler, including the metrics or models used to decide both when to scale or how many resources are necessary. Each auto-scaling approach has been designed with particular goals, focusing on several application architectures or cloud providers offering different scaling capabilities. The most differentiating factor is the final goal/evaluation criteria used for a particular auto-scaler: the prediction accuracy, the compliance with the SLA (defined in many ways such as response time or availability of resources), or the cost of resources. It seems quite obvious that comparing auto-scaling approaches is not that straight-forward. For example, let us consider two proposals, one focused in short-term prediction of workloads with a clearly diurnal pattern, and another one that tries to comply with the SLA even in the case of sudden bursts of traffic. Both auto-scaling systems may work well for their target systems, but clearly the latter approach is more prone to cause over-provisioning. Still, it would be wrong to assume that this extra cost due to over-provisioning state is enough to prefer one auto-scaling system instead of the other. Then, the target goal of the auto-scaler cannot be used as the classification criterion.

A possible grouping for auto-scalers could be done using their anticipation capabilities, arranging them in two classes: *reactive* and *proactive*. The former implies that the system reacts to changes in the workload only when those changes have been detected, using the last values obtained from the set of monitored variables; consequently, as resource provisioning takes some time, the desired effect may arrive when it is too late. Proactive systems anticipate future demands and make decisions taking them into consideration. We have chosen not to use this as a classification criterion because sometimes it is not clear whether a particular approach is purely reactive or proactive.

We decided to adopt the underlying theory or technique used to build the auto-scaler as our classification criterion. This will help the reader to better

understand the basic concepts of a particular group or category, including their advantages and limitations. Most reviewed works can fit in one or more of these five groups (note that some works propose hybridizations of several techniques):

1. Threshold-based rules (rules)
2. Reinforcement learning (RL)
3. Queuing theory (QT)
4. Control theory (CT)
5. Time series analysis (TS)

Commercial cloud providers offer purely reactive auto-scaling using threshold-based rules. The scaling decisions are triggered based on some performance metrics and predefined thresholds. This approach has become rather popular due to its (apparent) simplicity: rule-based auto-scaler are easy to provide as a cloud service, and are also easy to set-up by clients. However, the effectiveness of rules under bursty workloads is questionable.

Time series analysis covers a wide range of methods to detect patterns and predict future values on sequences of data points. The accuracy in the forecast value (e.g. future number of requests or average CPU utilization) will depend on selecting the right technique and setting the parameters correctly, specially the history window and the prediction interval. Time-series analysis is the main enabler of proactive auto-scaling techniques.

There are two auto-scaling methods that rely on modeling the system in order to determine its future resource needs. This is the case of both queuing theory and control theory. Queuing theory has been largely applied to computing systems, in order to find the relationship between the jobs arriving and leaving a system. A simple approach consists in modeling each VM (or set of VMs) as a queue of requests in order to estimate different performance metrics such as the response time. A main limitation of QT models is that they are too rigid, and need to be recomputed when there are changes in either the application or the workload.

Control theory also relies on creating a model of the application. The aim is to define a (reactive or proactive) controller to automatically adjust the required resources to the application demands. The nature and performance of a controller highly depends on both the application model and the controller itself. As we will see, many researchers consider that this type of auto-scaling has a great potential, specially when combined with resource prediction.

Finally, the last of our categories contains proposals based on reinforcement learning. Similarly to control theory, RL tries to automate the scaling task, but without using any a-priori knowledge or model of the application. Instead, RL tries to learn the most suitable action for each particular state on-the-fly, with a trial-and-error approach. Although the absence of model and adaptability of the

technique might seem the most appealing for auto-scaling, the truth is that RL suffers from long learning phases. The time required by the method to converge to an optimal policy can be unfeasible long.

As defined in previous chapter, the auto-scaling process is mainly related to the analysis and planning phases of the MAPE loop. Some auto-scaling proposals focus on one of these phases, but most of them cover both of them. Queuing theory and time series analysis are useful in the analysis phase in order to estimate the current utilization or future needs of the application. Threshold-based rules, reinforcement learning and control theory can be used in the planning phase to decide the scaling action, and they can be combined with a previous analysis phase involving for example, time series analysis.

Each subsequent section focuses on a taxonomy category, including the definition, methodologies, pros and cons. After that, we analyze a collection of papers fitting into the category, also discussing their features and limitations. This information is complemented with a set of tables summarizing the reviewed literature, one table per category. Each table row includes a synopsis of a reviewed auto-scaling paper, which includes:

1. The specific technique or combination of techniques applied
2. Whether an horizontal (H) or vertical (V) scaling is performed
3. The reactive (R) or proactive (P) nature of the approach
4. The performance metric considered (e.g. CPU load, input request rate)
5. The monitoring tool and the granularity or monitoring interval
6. Characteristics of the environment used to test the technique, including
    - The SLA considered for the application
    - The type of workload (either real or synthetic)
    - The experimental platform (simulator, custom testbed or real provider), together with the application benchmark

Note that many table entries contain a "-". This means that the reviewed paper does not provide enough information to know or infer the corresponding piece of information. Unfortunately, this happens more often than desirable. Also, note that authors have used many different mechanisms to assess the goodness of their auto-scaler, ranging from completely synthetic simulated environments to actual production systems.

5.2 Threshold-based Rules

Threshold-based auto-scaling rules or policies are very popular among cloud providers such as Amazon EC2, and third-party tools like RightScale [85]. The simplicity and intuitive nature of these policies make them very appealing to cloud clients. However, setting the corresponding thresholds is a per-application task, and requires a deep understanding of workload trends.

5.2.1 Description of the Technique

From the MAPE loop (see Section 2.3.2), rules are purely a decision-making technique (planning phase). The number of VMs or the amount of resources assigned to the target application will vary according to a set of rules, typically two: one for scaling up/out and one for scaling down/in. Rules are structured like these examples:

$$\begin{aligned} &\text{if } x_1 > thrU_1 \text{ and/or } x_2 > thrU_2 \text{ and/or } \ldots \\ &\quad \text{for } durU \text{ seconds then} \\ &\quad\quad n = n + s \text{ and} \\ &\quad\quad \text{do nothing for } inU \text{ seconds} \end{aligned} \tag{5.1}$$

$$\begin{aligned} &\text{if } x_1 < thrL_1 \text{ and/or } x_2 < thrL_2 \text{ and/or } \ldots \\ &\quad \text{for } durL \text{ seconds then} \\ &\quad\quad n = n - s \text{ and} \\ &\quad\quad \text{do nothing for } inL \text{ seconds} \end{aligned} \tag{5.2}$$

Each rule has two parts: the condition and the action to be executed when the condition is met. The condition part uses one or more performance metrics $x_1, x_2, ...$, such as the input request rate, CPU load or average response time. Each performance metric has upper $thrU$ and lower $thrL$ thresholds. If the condition is met for a given time ($durU$ or $durL$), then the corresponding action will be triggered. For horizontal scaling, the application manager should define a fixed amount s of VMs to be acquired or released, while for vertical scaling s refers to an increase or decrease of resources such as CPU or RAM. After executing an action, the auto-scaler inhibits itself for a small *cooldown* period defined by inU or inL.

The best way to understand threshold-based rules is by means of an example: *add 2 small instances when the average CPU usage is above 70% for more than 5 minutes, and then, do nothing for 10 minutes.*

5.2.2 Review of Proposals

Threshold-based rules constitute an easy to deploy and use mechanism to manage the amount of resources assigned to an application hosted in a cloud platform, dynamically adapting those resources to the input demand (e.g. [93], [87], [92], [89] [86], [88]). However, creating the rules requires an effort from the application manager (the client), who needs to select the suitable performance metric or logical combination of metrics, and also the values of several parameters, mainly thresholds. The experiments carried out by [87] show that application-specific metrics (e.g. the average waiting time in queue), obtain better performance that system-specific metrics (e.g. CPU load). Application managers also need to set the corresponding upper (e.g. 70%) and lower (e.g. 30%) thresholds for the performance variable (e.g. CPU load). Thresholds are the key for the correct working of the rules. In particular, Dutreilh et al. [93] remark that thresholds need to be carefully tuned in order to avoid oscillations in the system (e.g. in the number of VMs or in the amount of CPU assigned). To prevent this problem, it is advisable to set an *inertia*, *cooldown* or *calm* period, a time during which no scaling decisions can be committed, once an scaling action has been carried out.

Most authors and cloud providers use only two thresholds per performance metric. However, Hasan et al. [88] have considered using a set of four thresholds and two durations: *ThrU*, the upper threshold; *ThrbU*, which is slightly below the upper threshold; *ThrL*, the lower threshold; *ThroL*, which is slightly above the lower threshold. Used in combination, it is possible to determine the trend of the performance metric (e.g. trending up or down), and then perform finer auto-scaling decisions.

Conditions in the rules are usually based on a single, or at most two performance metrics, being the most popular the average CPU load of the VMs, the response time, or the input request rate. Both Dutreilh et al. [93] and Han et al. [86] use the average response time of the application. On the contrary, Hasan et al. [88] prefer using performance metrics from multiple domains (compute, storage and network) or even a correlation of several of them.

Note that in most cases the rules use the metrics directly as obtained from the monitor, thus acting in a purely reactive way. However, it is possible to carry out a previous analysis of the monitored data in order to predict the future (expected) behavior of the system and execute rules in a proactive way [94]. This topic will be discussed later, in the section devoted to time series analysis.

RightScale's auto-scaling algorithm [95] propose combining regular reactive rules with a voting process. If a majority of the VMs agree on that they should scale up/out or down/in, that action is taken; otherwise, no action is planned. Each VM votes to scale up/out or down/in based on a set of rules evaluated individually. After each scaling action, RightScale recommends a 15 minute-period of calm time because new machines generally take between 5 to 10 minutes to

Table 5.1: Summary of the reviewed literature about threshold-based rules. Table rows are as follow. (1) The reference to the reviewed paper. (2) A short description of the proposed technique. (3) The type of auto-scaling: horizontal (H) or vertical (V). (4) The reactive (R) and/or proactive (P) nature of the proposal. (5) The performance metric or metrics driving auto-scaling. (6) The monitoring tool used to gather the metrics. The remaining three fields are related to the environment in which the technique is tested. (7) The metric used to verify SLA compliance. (8) The workload applied to the application managed by the auto-scaler. (9) The platform on which the technique is tested.

Ref	Auto-scaling Techniques	H/V	R/P	Metric	Monitoring	SLA	Workloads	Experimental Platform
[86]	Rules	Both	R	CPU, memory, I/O	Custom tool. 1 minute	Response time	Synthetic. Browsing and ordering behavior of customers.	Custom testbed (called IC Cloud) + TPC
[87]	Rules	H	R	Average waiting time in queue, CPU load	Custom tool.	-	Synthetic	Public cloud. FutureGrid, Eucalyptus India cluster
[88]	Rules	Both	R	CPU load, response time, network link load, jitter and delay.	-	-	Only algorithm is described, no experimentation is carried out.	
[89]	Rules + QT	H	P	Request rate	Amazon CloudWatch. 1-5 minutes	Response time	Real. Wikipedia traces	Real provider. Amazon EC2 + Httperf + MediaWiki
[90]	RightScale + MA to performance metric	H	R	Number of active sessions	Custom tool	-	Synthetic. Different number of HTTP clients	Custom testbed. Xen + custom collaborative web application
[48]	RightScale + TS: LR and AR(1)	H	R/P	Request rate, CPU load	Simulated.	-	Synthetic. Three traffic patterns: weekly oscillation, large spike and random	Custom simulator, tuned after some real experiments.
[52]	RightScale	H	R	CPU load	Amazon CloudWatch	-	Real. World Cup 98	Real provider. Amazon EC2 + RightScale (PaaS) + a simple web application
[91]	RightScale + Strategy-tree	H	R	Number of sessions, CPU idle	Custom tool. 4 minutes.	-	Real. World Cup 98	Real provider. Amazon EC2 + RightScale (PaaS) + a simple web application.
[92]	Rules	V	R	CPU load, memory, bandwidth, storage	Simulated.	-	Synthetic	Custom simulator, plus Java rule engine Drools
[6]	Rules	V	R	CPU load	Simulated. 1 minute	Response time	Real. ClarkNet	Custom simulator

be operational. This auto-scaling technique has been adopted by several authors ([48], [96], [52], [91]). Chieu et al. [96] initially proposed a set of reactive rules based on the number of active sessions, but this work was extended in Chieu et al. [90] following the RightScale approach: if all VMs have active sessions above the given upper threshold, a new VM is provisioned; if there are VMs with active sessions below a given lower threshold and with at least one VM that has no active session, the idle one will be shut down.

As RightScale's voting system is based on rules, it inherits their main disadvantage: the technique is highly dependent on manager-defined threshold values, and also, on the characteristics of the input workload. This was the conclusion reached by Kupferman et al. [48] after comparing RightScale with other algorithms. Simmons et al. [91] try to overcome these problems with a strategy-tree, a tool that evaluates the deployed policy set, and switches among alternative strategies over time, in a hierarchical manner. Authors created three different scaling policies, customized to different input workloads, and the strategy-tree would switch among them based on the workload trend (analyzed with a regression-based technique).

In order to save costs, Kupferman et al. [48] (and other authors [89]) came up with an idea called *smart kill*. Many IaaS providers charge partial utilization hours as full hours. Therefore, it is advisable not to terminate a VM before the hour is over, even if the load is low. Apart from reducing costs, smart kill may also improve system performance: in case of an oscillating input workload, the costs of continuously shutting down and starting up VMs are avoided, improving boot-up delays and reducing SLA violations.

The popularity of rules as auto-scaling method is probably due to their simplicity and the fact that they are easy to understand for clients. However, this kind of technique shows two main problems: its reactive nature and the difficulty of selecting the correct set of performance metrics and the corresponding thresholds. The effectiveness of those thresholds is highly dependent on the workload changes, and may require frequent tuning. In order to solve the problem of static thresholds, Lorido-Botran et al. [6] introduce the concept of dynamic thresholds: initial values are set-up, but they are automatically modified as a consequence of the observed SLA violations. Meta-rules are included to define how the threshold used in the scaling rules may change to better adapt to the workload.

In conclusion, Rules can be used to easily automate the auto-scaling of a particular application without much effort, specially in the case of applications with quite regular, predictable patterns. However, in case of bursty workloads the client should consider a more advanced and powerful auto-scaling system from the rest of categories.

The main characteristics of the auto-scaling proposals based on rules and reviewed in this section are summarized in Table 5.1.

5.3 Reinforcement Learning

Reinforcement Learning (RL) [97] is a type of automatic decision-making approach that has been used by several authors to implement auto-scalers. Without any *a priori* knowledge, RL techniques are able to determine the best scaling action to take for every application state, given the input workload.

5.3.1 Description of the Technique

Reinforcement learning [97] focuses on learning through direct interaction between an agent (e.g. the auto-scaler) and its environment (e.g. the application as defined in Section 2.3.1). The auto-scaler will learn from experience (trial-and-error method) the best scaling *action* to take, depending on the current *state*, given by the input workload, performance or other set of variables. After executing an action, the auto-scaler gets a response or *reward* (e.g. performance improvement) from the system, about how good that action was. So, the auto-scaler will tend to execute actions that yield a high reward (best actions are *reinforced*). From now on, in this section the general term *agent* will be used, instead of *auto-scaler*.

The objective of the agent is to find a policy π that maps every state s to the best action a the agent should choose. The agent has to maximize the expected *discounted rewards* obtained in the long run:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots = \sum_{0}^{\infty} \gamma^k r_{t+k+1} \tag{5.3}$$

where r_{t+1} is the reward obtained at time $t+1$, and *gamma* is the discount factor.

The policy is based on a value function $Q(s,a)$, usually called the Q-value function. Every $Q(s,a)$ value estimates the future cumulative rewards by executing an action a in a state s. In other words, it represents the *goodness* of executing action a when in state s. The Q-value function can be defined as:

$$Q(s,a) = E_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a \right\} \tag{5.4}$$

There are many RL algorithms that can be used in order to obtain the Q-value function, but Q-learning is the most used in the literature. Typically, the $Q(s,a)$ values are stored in a lookup table, that maps all system states s to their best action a, and the corresponding Q-value. The Q-learning algorithm is sketched in Algorithm 2.

Step 4 of the algorithm involves choosing an action for a given state. Among the existing action selection policies, ϵ-greedy is often the one selected in the

1: Initialize the Q-values table, $Q(s, a)$.
2: Observe the current state, s.
3: **loop** {infinitely}
4: Choose an action, a, for state s based on one of the action selection policies, such as ϵ-greedy.
5: Execute the action, and observe the reward, r, as well as the new state, s'.
6: Update the Q-value for the state using the observed reward and the maximum reward possible for the next state. The resulting update formula is:

$$Q(s, a) = Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)] \tag{5.5}$$

7: Set the state s to the new state s'.
8: **end loop**

Algorithm 2: Q-learning basic steps

literature. Most of the time (with probability $1 - \epsilon$), the action with the best reward will be executed ($\arg_a \max Q(s, a)$); a random action will be chosen with a low probability ϵ, in order to explore non-visited actions. Once the action is executed, the corresponding Q-value is updated (step 6) with the obtained reward r and the maximum reward possible for the next state $\max_{a'} Q(s', a')$. The parameter γ is the discount factor that adjusts the importance given to future rewards. The update formula also includes a parameter α, that determines the learning rate. It can be the same for every state-action pair, or can be adjusted based on the number of times each state has been visited.

The policy learned by the agent is just the action that maximizes the Q-value in each state. Watkins and Dayan [98] proved that the discrete case of Q-learning converges to an optimal policy under certain conditions, and this is independent of the initial values of the Q-table. If each pair (s, a) is visited an infinite number of times, then the lookup table converges to a unique set of values $Q(s, a) = Q^*(s, a)$, which defines a stationary deterministic optimal policy. Note that the auto-scaling process is a continuing task, not stationary. For this reason, the Q-learning policy has to be learned infinitely and adapted to the workload changes.

An algorithm very similar to Q-learning is SARSA [97]. In contrast to Q-learning, it does not use the maximum reward for the next state $\max_{a'} Q(s', a')$ to update the $Q(s, a)$ value. Instead, the same action selection policy (check Step 4 in Algorithm 2) is applied to the new state s', in order to determine an action a'. Then, the corresponding $Q(s', a')$ value is used to update the current $Q(s, a)$, as shown in the following formula:

$$Q(s, a) = Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)] \tag{5.6}$$

The effect of a scaling decision takes some time to have an impact on the application, and the reward comes after a delay. Given that RL makes a decision with current information (application state) about a future reward (e.g. response time), a proactive nature is assumed for all RL approaches. The method includes two phases of the MAPE process: analyze and plan. First, information about application and rewards are collected in a lookup table (or any other structure) for later use (analyze phase). Then, in the planning phase, this information is used to decide the best scaling action.

5.3.2 Review of Proposals

Three basic elements have to be defined in order to apply RL to auto-scaling: the action set A, the state space S, and the reward function R. The first two highly depend on the type of scaling, either horizontal or vertical, whereas the reward function usually takes into account the cost of the acquired resources (renting VMs, bandwidth, ...) and the penalty cost for SLA violations. In case of horizontal scaling, the state is mostly defined in terms of the input workload and the number of VMs. For example, Tesauro et al. [101] propose using (w, u_{t-1}, u_t), where w is the total number of user requests observed per time period; u_t and u_{t-1} are the number of VMs allocated to the application in the current time step, and the previous time step, respectively; Dutreilh et al. [99] considered the (w, u, p) state definition, where p is the performance in terms of average response time to requests, bounded by a value P_{max} chosen from experimental observations. The set of actions for horizontal scaling are usually three: add a new VM, remove an existing VM, and do nothing.

In contrast, the state definition for vertical scaling takes into account the amount of resources assigned to each VM (CPU and memory, mostly). In particular, Rao et al. [102] [103] considered the following state definition: $(mem_1, time_1, vcpu_1, \ldots, mem_u, time_u, vcpu_u)$, where mem_i, $time_i$ and $vcpu_i$ are the ith VM's memory size, scheduler credit and the number of virtual CPUs, respectively. For each of the three variables, possible operations can be either increase, decrease or no-operation (i.e. maintain the previous value). The actions for the RL task are defined as the combinations of the operations on each variable. Given that the variables are continuous, operations are discretized: mem is reconfigured in blocks of 256MB; scheduler changes ($time$) in 256 credits; and $vcpu$ in 1 unit. The same authors propose in [104] a distributed approach, in which a RL agent is learned per VM. In this case, the state is configured as $(CPU, bandwidth, memory, swap)$.

Even though Q-learning is the most extended algorithm in auto-scaling, there are some exceptions: e.g. Tesauro et al. [101] use the SARSA approach, explained before. Some of the articles referenced in this section ([102], [104], [105], [103]) do not specify which is the specific RL algorithm applied in the experiments,

Table 5.2: Summary of the reviewed literature about reinforcement learning

Ref	Auto-scaling Techniques	H/V	R/P	Metric	Monitoring	SLA	Workloads	Experimental Platform
[93]	Rules + RL	H	P	Request rate, response time, number of VMs	Custom tool. 20 seconds	Response time	Synthetic. Made up of 5 sinusoidal oscillations	Custom testbed + RUBiS
[99]	RL	H	P	Number of user requests, number of VMs, response time	-	Response time, cost	Synthetic. With sinusoidal pattern	Custom testbed. Olio application + Custom decision agent (VirtRL)
[100]	RL	H	P	Number of user requests, number of VMs, time	Simulated	Response time, cost	Synthetic. Based on Poisson distribution	Custom simulator (Matlab)
[101]	RL(+ANN) - SARSA + Queuing model	H	P	Arrival rate, previous number of VMs	-	Response time	Synthetic. Poisson distribution (open-loop), different number of users and exponentially distributed think times (closed-loop)	Custom testbed (shared data center). Trade3 application (a realistic simulation of an electronic trading platform)
[102]	RL(+ANN)	V	P	CPU and memory usage	Custom tool	Throughput, response time	Synthetic: 3 workload mixes (browsing, shopping and ordering)	Custom testbed. Xen + 3 applications (TPC-C, TPC-W, SpecWeb)
[103]	RL(+ANN)	V	P	CPU and memory usage	Custom tool	Response time	Synthetic: 3 workload mixes (browsing, shopping and ordering)	Custom testbed. Xen + 3 applications (TPC-C, TPC-W, SpecWeb)
[104]	RL(+CMAC)	V	P	CPU, I/O, memory, swap	Custom tool (scripts)	Response time, throughput	Real. ClarkNet trace	Custom testbed. Xen + 3 applications (TPC-C, TPC-W, SpecWeb)
[105]	RL(+Simplex)	V	P	CPU, memory - application parameters	Custom tool	Response time, throughput	Synthetic. Different number of clients	Custom testbed. Xen + 2 applications (TPC-C, TPC-W)
[106]	CT - PID controller + RL + ARMAX model + SVM regression	V	P	Application adaptive parameters – CPU and memory	-	Application-related benefit function	Synthetic	Custom testbed. Xen + 2 real applications (Great Lake Forecasting System, Volume Rendering)

but the problem definition and update function resembles those of SARSA. Both Q-learning and SARSA present several problems, that have been addressed in a number of ways [93], [99] [100], [101], [102]:

- Bad initial performance and large training time. The performance obtained during live on-line training may be unacceptably poor, both initially and during a long training period, before a good enough solution is found. The algorithm needs to explore the different states and actions.
- Large state-space. This is often called the *curse of dimensionality problem*. The number of states grows exponentially with the number of state variables, which leads to scalability problems. In the simplest form, a lookup table is used to store a separate value for every possible state-action pair. As the size of such a table increases, any access to the table will take a longer time, delaying table updates and action selection.
- Exploration actions and adaptability to environment changes. Even assuming that an optimal policy is found, the environment conditions (e.g. workload pattern) may change and the policy has to be re-adapted. For this purpose, the RL algorithm executes a certain amount of exploration actions, believed to be suboptimal. This could lead to undesired bad performance, but it is essential in order to adapt the current policy. Following this method, RL algorithms are able to cope with relatively smooth changes in the behavior of the application, but not to sudden burst in the input workload.

The problem of the bad performance in the early steps has been addressed in a number of ways. Its main cause is the lack of initial information and the random nature of the exploration policy. For this reason, Dutreilh et al. [93] used a custom heuristic to guide the state space exploration, and the learning process lasted for 60 hours. The same authors [99] further propose an initial approximation of the Q-function that updates the value for all states at each iteration, and also speeds up the convergence to the optimal policy. Reducing this training time can also be addressed with a policy that visits several states at each step [102] or using parallel learning agents [100]. In the latter, each agent does not need to visit every state and action; instead, it can learn the value of non-visited states from neighboring agents. A radically different approach to avoid the poor early performance of on-line training consists of using an alternative model (e.g. a queuing model) to control the system, whilst the RL model is trained off-line on collected data [101].

Some authors have proposed methods to address the curse of dimensionality issue, reducing the state space. Bu et al. [105] use a Simplex optimization that selects the *promising* states that would return a high reward. A parallel

approach would also help coping with large state spaces. Barrett et al. [100] create an agent per VM, that keeps its own, small lookup table. Using a lookup table for representing the Q-function is not efficient, and other nonlinear function approximators can be utilized, such as neural networks, CMACs (Cerebellar Model Articulation Controllers), regression trees, support vector machines, wavelets and regression splines. For example, neural networks [101], [102] take the state-action pairs as input and output the approximated Q-value. They are also able to predict the value for non-visited states, and deal with continuous state spaces. Rao et al. [104] combine the parallel approach (an agent per VM) with a CMAC [107] [104] to represent the Q function. Authors found that updates of the CMAC-based Q table only need 6.5 milliseconds, in comparison with the 50-second update time in a neural network.

It is also worth mentioning the usefulness of RL in other tasks that can be tightly linked to the auto-scaling problem. For example, application parameter configuration [103] [105]. Xu et al. [103] use an ANN-based RL agent to tune parameters directly related to the application and VM performance, such as MaxClients, Keepalive timeout, MinSpareServers, MaxThreads and Session timeout (these are examples of tunable parameters from Tomcat or Apache applications).

RL can be used in combination with other methods such as control theory. Following a radically different approach, Zhu and Agrawal [106] combine a PID controller with an RL agent in charge of estimating the derivative term (this is further explained in Section 5.5). The controller guides the parameter adaptation of applications (e.g. image size) in order to meet the SLA. Then, virtual resources, CPU and memory, are dynamically provisioned according to the change in the adaptive parameters.

Before finishing this section, it is important to remark the interesting capability of RL algorithms to capture the best management policy for a target scenario, without any *a-priori* knowledge. The client does not need to define a particular model for the application; instead, it is learned online and adapted if the conditions of the application, workload or system change. In our opinion, RL techniques could be a promising approach to solve the auto-scaling task of general applications, but the field is not mature enough to satisfy the requirements of a real production scenario. In this open research field, efforts should be addressed towards providing an adequate adaptability to sudden bursts in the input workload, and also to deal with continuous state spaces and actions.

Table 5.2 shows a summary of the articles reviewed in this section.

5.4 Queuing Theory

Classical queuing theory has been extensively used to model Internet applications and traditional servers. Queuing theory can be used for the analysis phase of

the auto-scaling task in elastic applications (see Section 2.3.2), i.e., estimating performance metrics such as the queue length or the average waiting time for requests. This section describes the main characteristics of a queuing model and how they can be applied to scalable scenarios.

5.4.1 Description of the Technique

Queuing theory makes reference to the mathematical study of waiting lines, or queues. The basic structure of a model is depicted in Figure 5.1. Requests arrive to the system at a mean arrival rate λ, and are enqueued until they are processed. As the figure shows, one or more servers may be available in the model, that will attend requests at a mean service rate μ.

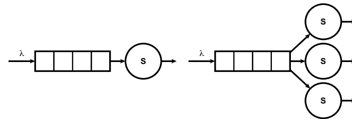


Figure 5.1: A simple queuing model with one server (left) and multiple servers (right)

Kendall's notation is the standard system used to describe and classify queuing models. A queue is denoted by $A/B/C/K/N/D$. This is the meaning of each element in the notation:

A**:** *Inter-arrival time distribution.*

B**:** *Service time distribution.*

C**:** *Number of servers.*

K**:** *System capacity* or *queue length*. It refers to the maximum number of customers allowed in the system including those in service. When the system is fully occupied, further arrivals are rejected.

N**:** *Calling population.* The size of the population from which the customers come. If the requests come from an infinite population of customers, the queuing model is *open*, whereas a *closed* model is based on a finite population of customers.

D**:** *Service discipline* or *priority order.* The service discipline or priority order in which jobs in the queue are served. The most typical one is FIFO/FCFS (First In First Out / First Come First Served), in which the requests are

served in the order they arrived. There are alternatives such as LIFO/LCFS (Last in First Out / Last Come First Served) and PS (Processor Sharing), among others.

Elements K, N and D are optional; if not present, it is assumed that $K = \infty$, $N = \infty$ and $D =$ FIFO. The most typical values for both inter-arrival time A and service time B, are M, D and G. M stands for Markovian and it refers to a Poisson process, which is characterized by a parameter λ that indicates the number of arrivals (requests) per time unit. Therefore, the inter-arrival or the service time will follow an exponential distribution. D means deterministic or constant times. Another commonly used value is G, that corresponds to a general distribution with known parameters.

The elastic application scenario described in Section 2.3.1 can be formulated using a simple queuing model, considering a single queue representing the load balancer, that distributes the requests among n VMs (see Figure 5.1). In order to represent more complex systems such as multi-tier applications, a *queuing network* can be utilized. For example, each tier can be modeled as a queue with one or n servers.

Queuing theory is used to analyze systems with a stationary nature, characterized by constant arrival and service rates. Its objective is to derive some performance metrics based on the queuing model and some known parameters (e.g. arrival rate λ). Examples of performance metrics are the average time waiting in the queue, and the mean response time. In case of scenarios with changing conditions (i.e. non-constant arrival and services rates), such as our target, scalable applications, the parameters of the queuing model have to be periodically recalculated, and the metrics recomputed.

There are two main ways to solve queuing models: analytically and by means of simulation. The former can only be used with simple models with well-defined distributions for arrival and service processes, such as M/M/1 and G/G/1. A well-known analytical way of solving queuing networks is Mean Value Analysis (MVA) [108]. When analytical approaches are not feasible, that is, for relatively complex models, simulation can still be used to obtain the desired metrics.

M/M/1 is the simplest queuing (Poisson-based) model, where both the arrival times and the service times follow exponential distributions. In this case, the mean response time R of a M/M/1 model can be calculated as $R = \frac{1}{\mu - \lambda}$, given a service rate μ and an arrival rate λ (the response time R is the sum of the enqueued time and the service time). Another simple queuing model is G/G/1, in which the system's inter-arrival and service times are governed by general distributions with known mean and variance. The behavior of a G/G/1 system can be captured using the following equation: $\lambda \geq \left[s + \frac{\sigma_a^2 + \sigma_b^2}{2(R-s)} \right]^{-1}$, where R is the mean response time, s is the average service time for a request and σ_a^2, σ_b^2 are

the variances of inter-arrival time service time, respectively.

In many queuing scenarios, it is useful to apply *Little's Law*, which states that the average number of customers (or requests) $E[C]$ in the system is equal to the average customer arrival rate λ multiplied by the average time of each customer in the system $E[T]$: $E[C] = \lambda \times E[T]$.

5.4.2 Review of Proposals

In the literature, both simple queuing models and more complex queuing networks have been widely used to analyze the performance of computer applications and systems.

Ali-Eldin et al. [114] [115] model a cloud-hosted application as a G/G/n queue, in which the number of servers n is variable. The model can be solved to compute, for example, the necessary resources required to process a given input workload λ, or the mean response time for requests, given a configuration of servers.

Queuing networks can also be used to model elastic applications, representing each VM (server) as a separate queue. For example, Urgaonkar et al. [109] use a network of G/G/1 queues (one per server). They use histograms to predict the peak workload. Based on this value and the queuing model, the number of servers per application tier is calculated. This number can be corrected using reactive methods. The clear drawback is that provisioning for peak load drives to high under-utilization of resources.

Multi-tier applications can be studied using one or more queues per tier. Zhang et al. [111] considered a limited number of users, and thus, they used a closed system with a network of queues; this model can be efficiently solved using MVA. Han et al. [112] modeled a multi-tier application as an open system with a network of G/G/n queues (one per each tier); this model is used to estimate the number of VMs that need to be added to, or removed from, the bottleneck tier, and the associated cost (VM usage is charged per minute, instead of the typical per-hour cost model). Finally, Bacigalupo et al. [113] considered a queuing network of three tiers, solving each tier to compute the mean response time.

The discussed approaches are used as part of the analysis phase of the MAPE loop. Many different techniques can be used to implement the planning phase, such as a predictive controller [115] or an optimization algorithm (e.g. distributing servers among different applications, while maximizing the revenue [110]). The information required for a queuing model, such as the input workload (number of requests) or service time can be obtained by on-line monitoring [112] [109] or estimated using different methods. For example, Zhang et al. [111] used a regression-based approximation in order to estimate the CPU demand, based on the number and type (browsing, ordering or shopping) of client requests.

Queuing models have been extensively used to model applications and sys-

Table 5.3: Summary of the reviewed literature about queuing theory

Ref	Auto-scaling Techniques	H/V	R/P	Metric	Monitoring	SLA	Workloads	Experimental Platform
[109]	QT + Histogram + Thresholds	H	R/P	Peak workload	Custom tool. 15 minutes	Response time	Synthetic and Real (World Cup 98)	Custom testbed. Xen + 2 applications (RUBiS and RUBBOS)
[110]	QT	H	R	Arrival rate, service time	Simulated	Response time	Real. E-commerce website (2001) + Synthetic traces	Custom simulator (Monte-Carlo)
[111]	QT + Regression (Predict CPU load)	-	P	Number and type of transactions (requests), CPU load	Custom tool. 1 minute	-	Synthetic (browsing, ordering and shopping)	Custom simulator, based on C++Sim. + Data collected from TPC-W
[112]	QT (model) + Reactive scaling	H	R	Arrival rate, service time	Custom tool. 1 minute	Response time, cost	Synthetic (browsing, ordering and shopping)	Custom testbed (called IC Cloud) + TPC-W benchmark
[113]	QT + Historical performance model	H	P	Arrival rate	-	Response time	Synthetic (browsing, buying)	Custom testbed. Eucalyptus + IBM Performance Benchmark Sample Trade

tems. They usually impose a fixed architecture, and any change in structure or parameters require solving the model (with analytical or simulation-based tools). For this reason, they are not cheap when used with elastic (dynamically variable) applications that have to deal with a changing input workload and a varying pool of resources. Additionally, a queuing system is an analysis tool, that requires additional components to implement a complete auto-scaler. Queuing models could be useful for some particular cases of applications, e.g. when the relationship between the number of requests and needed resources is quite linear. Although there are efforts to model more complex multi-tier applications, queuing theory might not be the best option when trying to design a general-purpose auto-scaling system.

Table 5.3 contains a summary of the articles reviewed in this section.

5.5 Control Theory

Control theory has been applied to automate the management of different information processing systems, such as web server systems, storage systems and data centers/server clusters. For cloud-hosted, elastic applications, a control system may combine both phases of the auto-scaling task (analysis and planning).

5.5.1 Description of the Technique

The main objective of a controller is to automate the management (e.g. scaling task) of a target system (e.g. a cloud application as defined in Section 2.3.1). The controller has to maintain the value of a *controlled variable* y (e.g. CPU load), close to the desired level or *set point* y_{ref}, by adjusting the *manipulated variable* u (e.g. number of VMs). The manipulated variable is the input to the target system, whereas the controlled variable is measured by a sensor and considered the output of the system.

There are three main types of control systems: open loop, feedback and feed-forward. *Open-loop* controllers, also referred to as non-feedback, compute the input to the target system using only the current state and its model of the system. They do not use feedback to determine whether the system output y has achieved the desired goal y_{ref}. In contrast, *feedback* controllers observe system output, and are able to correct any deviation from the desired value (see Figure 5.2). *Feed-forward* controllers try to anticipate to errors in the output. They predict, using a model, the behavior of the system, and react before the error actually occurs. The prediction may fail and, for this reason, feedback and feed-forward controllers are usually combined.

From now on, focus will be put on feedback controllers, as they are frequently used in the literature. They can be classified into several categories [116]:

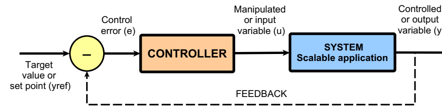


Figure 5.2: Block diagram of a feedback control system

Fixed gain controllers: This class of controllers are very popular due to their simplicity. However, after selecting the tuning parameters, they remain fixed during the operation time of the controller. The most common controller in this class is called Proportional Integral Derivate (PID), with the following control algorithm:

$$u_k = K_p e_k + K_i \sum_{j=1}^{k} e_j + K_d(e_k - e_{k-1}) \tag{5.7}$$

where u_k is the new value for the manipulated variable (e.g. new number of VM); e_k is the difference between the output y_k and the set point y_{ref}; and K_p, K_i and K_d are the proportional, integral and derivative gain parameters (respectively) that need to be adapted to a given target system. Different variants of the PID controller are used, such as the Proportional Integral (PI) controller or the Integral (I) controller. The latter can be represented as $u_k = u_{k-1} + K_i e_k$

Adaptive controllers: As the name suggests, adaptive control is able to adjust the parameter values on-line, in order to adapt the controller to the changing conditions of the environment. Examples of adaptive controllers are self-tuning PID controllers, gain-scheduling and self-tuning regulators. They are suitable for slowly varying workload conditions, but not for sudden bursts; in that case, the on-line model estimation process may fail to capture the dynamics of the system.

Model predictive controllers (MPC): MPCs follow a proactive approach; the future behavior (output) of the system is predicted, based on the model and the current output. In order to maintain this output close to the target value, the controller solves an optimization problem taking into account a pre-defined cost function. An example of MPC is the look-ahead controller [117].

As explained before, the controller has to adjust the input variable (e.g. number of VMs), in order to maintain the desired value in the output variable (e.g. average

CPU load of 90%). For this purpose, a formal relationship between the input and the output has to be modeled, so as to determine how a change in the former affects the value of the output. This formal relationship is denoted as *transfer function* in classical control theory, *state-space function* in modern control theory, or simply as *performance model*. PID controllers consider a simple linear equation, but there are many alternatives that consider non-linear approaches, and even several input and output variables (yielding a *Multiple-Input Multiple-Output* (MIMO) controller, instead being *Single-Input Single-Output* (SISO)). In the literature, authors have proposed different performance models of the system, including these:

***ARMA(X)* [121]:** An ARMA (auto-regressive moving average) model is able to capture the characteristics of a time series and then makes predictions of future values. ARMAX (ARMA with eXogenous input) models the relationship between two time series. Both models are further explained in Section 5.6.

***Kalman filter* [127]:** It is a recursive algorithm for making predictions based on time series.

***Smoothing splines* [122]:** It is a method of smoothing (i.e fitting a smooth curve to a set of noisy observations) using splines. This term refers to a polynomial function, defined by multiple subfunctions.

***Kriging model or Gaussian Process Regression* [123]:** It extends traditional linear regression with a statistical framework that is able to predict the value of the target function in un-sampled locations together with a confidence measure.

***Fuzzy model* [124] [125] [126]:** Fuzzy models are based on fuzzy rules. They are based on the idea that the membership of an element to a set has a degree value in a continuous interval between 0 and 1 (in contrast to Boolean logic). Fuzzy models are further described in Subsection 5.5.2.

5.5.2 Review of Proposals

Fixed-gain controllers, including PID, PI and I, are the simplest controller types, and have been widely used in the literature. For example, Lim et al. [119] [120] use an I controller to adjust the number of VMs based on average CPU usage, while Park and Humphrey [118] apply a PI controller to manage the resources required by batch jobs, based on their execution progress. Gain parameters K_p and K_I can be set manually, based on trial-and-error [119] or using an application-specific model. For example, in [118] a model is constructed to estimate the progress of a

job with respect to the resources provisioned. Zhu and Agrawal [106] rely on a RL agent in order to estimate the derivative term of a PID controller. With the trial-and-error training, the RL agent learns to minimize the sum of the squared error of the control variables (the adaptive parameters) without violating the time and budget constraints over time.

Adaptive control techniques are also rather popular. For example, Ali-Eldin et al. [115] propose combining two proactive, adaptive controllers for scaling down, using dynamic gain parameters based on input workload. A reactive approach is used for scaling up. The same authors propose in [114] an adaptive, proportional controller, using a proactive approach for both scaling up and down, and taking into account the VM startup time. As stated before, adaptive control techniques rely on the use of performance models. Padala et al. [121] propose a MIMO adaptive controller that uses a second-order ARMA to model the non-linear and time-varying relationship between the resource allocation and its normalized performance. The controller is able to adjust the CPU and disk I/O usage. Bodík et al. [122] combine smoothing splines (used to map the workload and number of servers to the application performance) with a gain-scheduling adaptive controller. Kalyvianaki et al. [127] designed different SISO and MIMO controllers to determine the CPU allocation of VMs, relying on Kalman filters, whereas [123] utilized a Kriging model to predict job completion time as a function of the number of VM, the number of incoming requests and the jobs enqueued at the master node.

Fuzzy models have been used as a performance model in control systems to relate the workload (input variable) and the required resources (output variable). First, both input and output variables of the system are mapped into fuzzy sets. This mapping is defined by a membership function that determines a value within the interval [0,1]. A fuzzy model is based on a set of rules, that relate the input variables (pre-condition of the rule), to the output variables (consequence of the rule). The process of translating input values into one or more fuzzy sets is called fuzzification. Defuzzification is the inverse transformation which derives a single numeric value that best represents the inferred fuzzy values of the output variables. A control system that relies on a rule-based fuzzy model is called a *fuzzy controller*. Typically, the rule set and membership functions of a fuzzy model are fixed at design time, and thus, the controller is unable to adapt to a highly dynamic workload. An adaptive approach can be used, in which the fuzzy model is repeatedly updated based on on-line monitored information [124], [125]. Xu et al. [124] applied an adaptive fuzzy controller to the business-logic tier of a web application, and estimated the required CPU load for the input workload. A similar approach is followed by [125], but here authors focus on the database tier. They claim that they use the fuzzy model to predict the future resource needs; however, they use the workload of the current time step t, to calculate

the resource needs r_{t+1} of the time step $t+1$, based on the assumption that no sudden change happened within the duration of a time step.

A further improvement is the *neural fuzzy controller*, which uses a four-layer neural network (see Section 5.6) to represent the fuzzy model. Each node in the first layer corresponds to one input variable. The second layer determines the membership of each input variable to the fuzzy set (the fuzzification process). Each node in layer 3 represents the precondition part of one fuzzy logic rule. An finally, the output layer acts as a defuzzifier, which converts fuzzy conclusions from layer 3 into numeric output in terms of resource adjustment. At early steps, the neural network only contains the input and output layers. The membership and the rule nodes are generated dynamically through the structure and parameters learning. Lama and Zhou [126] relied on a neural fuzzy controller, that is capable of self-constructing its structure (both the fuzzy rules and the membership functions) and adapting its parameters through fast on-line learning (a reconfiguring controller type).

Finally, the last type of controllers that follow a proactive approach are MPCs. Roy et al. [117] combined an ARMA model for workload forecasting, with the look-ahead controller in order to optimize the resource allocation problem. Fuzzy models can also be used to create a Fuzzy Model Predictive Controller [128].

The suitability of controllers for the auto-scaling task highly depends on the type of controller and the dynamics of the target system. The idea of having a controller that automates the process of adding/removing resources is very appealing, but still, the problem is how to create a reliable performance model that maps the input and output variables. Simple reactive controllers could be used for applications with easy to predict needs. However, it seems advisable to focus efforts towards both adaptive and MPCs that are able to adapt the application model and could be more suitable to produce a general auto-scaling solution.

Table 5.4 shows a summary of the articles reviewed in this section.

5.6 Time Series Analysis

Time series are used in many domains including finance, engineering, economics and bioinformatics, generally to represent the change of a measurement over time. A time series is a sequence of data points, measured typically at successive time instants spaced at uniform time intervals. An example is the number of requests that reaches an application, taken at one-minute intervals. The time series analysis can be used in the analysis phase of the auto-scaling process, in order to find repeating patterns in the input workload or to try to forecast future values.

5.6.1 Description of the Technique

Given the scenario described in Section 2.3.1, a certain performance metric, such as average CPU load or the input workload, will be sampled periodically, at fixed intervals (e.g. each minute). The result will be a time series X containing a sequence of w observations (w is the time series length):

$$X = x_t, x_{t-1}, x_{t-2}, ..., x_{t-w+1} \tag{5.8}$$

Time series techniques can be applied in order to predict future values of the metric, e.g. the future workload or resource usage. Based on this predicted value, a suitable auto-scaling action can be planned, using for example a set of predefined rules [94], or solving an optimization problem for the resource allocation [117].

Formally, the objective of time series analysis is to forecast future values of the time series, based on the last q observations, which is denoted as *input window* or *history window* (where $q \leq w$). The future value $\hat{x}_{t+r}$ is r intervals ahead of the input window. Time series analysis techniques can be classified into two broad groups: some of them focus on the direct prediction of future values, whereas other techniques try to identify the pattern (if present) followed by the time series, and then extrapolate it to predict future values. The first group includes moving average, auto-regression, ARMA (combining both), exponential smoothing and different approaches based on machine learning:

***Moving average methods (MA)*:** They can be used to smooth a time series in order to remove noise or to make predictions. The forecast value $\hat{x}_{t+r}$ is calculated as the weighted average of the last q consecutive values. Typically, the prediction interval r is set to 1. Then, the general formula is as follows: $\hat{x}_{t+r} = a_1 x_t + a_2 x_{t-1} + \ldots + a_q x_{t-q+1}$, where $a_1, a_2, \ldots, a_q$ are a set of positive weighting factors that must sum 1. Simple moving average MA(q) considers the arithmetic mean of the last q values, i.e., it assigns equal weight $\frac{1}{q}$ to all observations. In contrast, the weighted moving average WMA(q) assigns a different weight to each observation. Typically, more weight is given to the most recent terms in the time series, and less weight to older data.

***Exponential smoothing (ES)*:** Similarly to moving average, it calculates the weighted average of past observations, but exponential smoothing takes into account all the past history of the time series (w observations). It assigns *exponentially* decreasing weights over time. A new parameter is introduced, a smoothing factor α that weakens the influence of past data. There are different versions of exponential smoothing such as *simple* or single ES, and *Brown's double ES* [129]. In simple ES, the following formula is used recursively to calculate the current smoothed value s_t, based on the current

observation x_t and the previous smoothed value s_{t-1}: $s_t = \alpha x_t + (1-\alpha)s_{t-1}$. The forecast for the next period $\hat{x}_{t+1}$ is simply the current smoothed value s_t. The predictor formula for simple exponential smoothing can be expanded as follows:

$$\begin{aligned}
\hat{x}_{t+1} =& \alpha x_t + (1-\alpha)\hat{x}_t \\
=& \alpha x_t + (1-\alpha)[\alpha x_{t-1} + (1-\alpha)\hat{x}_{t-1}] \\
=& \alpha x_t + \alpha(1-\alpha)x_{t-1} + \\
& (1-\alpha)^2[\alpha x_{t-2} + (1-\alpha)\hat{x}_{t-2}] \\
& \vdots \\
=& \alpha x_t + \alpha(1-\alpha)x_{t-1} + \alpha(1-\alpha)^2 x_{t-2} + \\
& \ldots + (1-\alpha)^{w-1}\hat{x}_{t-w+1}
\end{aligned} \tag{5.9}$$

where $\hat{x}_{t+1}$ represents the forecast value for the period $t+1$, based on actual x_t value and the previous values of the time series. $\hat{x}_t$ refers to the forecast made for period t. The first smoothed value $\hat{x}_{t-w+1}$ can be set to the initial value of the time series x_{t-w+1}, or to the mean of the few first observations.

Simple ES is suitable for time series that have no significant trend changes. For time series with an existing linear trend, the Brown's double ES should be applied. It calculates two smoothing series:

$$\begin{aligned}
s_t^1 &= \alpha x_t + (1-\alpha)s_{t-1}^1 \\
s_t^2 &= \alpha s_t^1 + (1-\alpha)s_{t-1}^2
\end{aligned} \tag{5.10}$$

The second smooth series s_t^2 is obtained by applying simple ES to series s_t^1. Both smoothed values s_t^1 and s_t^2 are used to estimate the level c_t and trend d_t of the time series at the time t. Based on these, the forecast value $\hat{x}_{t+r}$ is calculated as follows:

$$\begin{aligned}
\hat{x}_{t+r} &= c_t + r d_t \\
c_t &= 2s_t^1 - s_t^2 \\
d_t &= \frac{\alpha}{1-\alpha}s_t^1 - s_t^2
\end{aligned} \tag{5.11}$$

Auto-regression **of order p, AR(p):** The prediction formula is determined as the linear wighted sum of the p previous terms in the series: $\hat{x}_{t+1} = b_1 x_t + b_2 x_{t-1} + \ldots + b_p x_{t-p+1} + \epsilon_t$. Parameter p corresponds to the number of terms in the AR equation, that may be different from the history window

length w. The formula may include a white noise term ϵ_t. The key is to derive the best values for the weights or auto-regression coefficients b_1, $b_2, \ldots, b_p$. There are a diversity of techniques for computing AR coefficients, such as least squares or the maximum likelihood method. In the literature, the most common method is based on the calculation of auto-correlation coefficients and the Yule-Walker equations. The auto-correlation function (ACF) of a time series gives correlations between x_t and x_{t-k} for lag $k = 1, 2, 3, \ldots$:

$$r_k = \frac{covariance(x_t, x_{t-k})}{var(x_t)} = \frac{E[(x_t - \mu)(x_{t-k} - \mu)]}{var(x_t)} \tag{5.12}$$

The autocorrelation can be estimated as:

$$r_k = \frac{1}{(w-k)\sigma^2} \sum_{t=1}^{w-k} (x_t - \bar{x}) * (x_{t-k} - \bar{x}) \tag{5.13}$$

The full autocorrelation function can be derived by recursively calculating $r_k = \sum_{i=1}^{p} b_k r_{i-k}$. The result is a set of linear equations called the Yule-Walker equations, that can be represented in matrix form as:

$$\begin{bmatrix} 1 & r_1 & r_2 & r_3 & \ldots & r_{p-1} \\ r_1 & 1 & r_1 & r_2 & \ldots & r_{p-2} \\ r_2 & r_1 & 1 & r_1 & \ldots & r_{p-3} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ r_{p-1} & r_{p-2} & r_{p-3} & r_{p-4} & \ldots & 1 \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_p \end{bmatrix} = \begin{bmatrix} r_1 \\ r_2 \\ r_3 \\ \vdots \\ r_p \end{bmatrix} \tag{5.14}$$

By solving this set of equations, auto-regression coefficients $b_1, b_2, \ldots, b_p$ can be derived for any p value. For example, for AR(1) (with $p = 1$), the auto-regression coefficient b_1 is equal to the corresponding autocorrelation coefficient r_1 ($b_1 = r_1$).

***Auto-Regressive Moving Average*, ARMA(p, q):** This model combines both auto-regression (of order p) and moving average (of order q). As stated before, AR takes into account the last p observations in the time series x_t, x_{t-1},
$x_{t-2}, \ldots, x_{t-p}$. The MA model, which is different from the MA method described previously, is the sum of the time series mean μ, plus the innovations or white error terms $\epsilon_t, \epsilon_{t-1}, \ldots, \epsilon_{t-q}$.

$$x_t = \mu + \epsilon_t + a_1\epsilon_{t-1} + a_2\epsilon_{t-2} + \ldots + a_q\epsilon_{t-q} \tag{5.15}$$

where $\epsilon_t \sim N(0, \sigma^2)$. Then, the ARMA model is represented as:

$$x_t = b_1 x_{t-1} + \ldots + b_p x_{t-p} + \epsilon_t + a_1 \epsilon_{t-1} + \ldots + a_q \epsilon_{t-q} \quad (5.16)$$

Note that an ARMA$(0, q)$ is a pure MA model; and an ARMA$(p, 0)$ corresponds to an AR model. There are several techniques for selecting the appropriate values for the orders p and q, and estimating the coefficients $a_1, a_2, \ldots, a_q$ and $b_1, b_2, \ldots, b_p$.

ARMA is a suitable model for stationary processes, i.e. the mean and variance of the time series remain constant over time. Thus, the time series must not show any trend (the variations of the mean) or seasonal variations. If the AR model correctly fits the time series, the residual ϵ is a white noise that shows no pattern. An extension of the ARMA model, called ARIMA (Auto-Regressive Integrated Moving Average), can be applied to non-stationary time series. Another extension called ARMAX(p, q, b), or ARMA with eXogenous inputs, is able to capture the relationship between a given time series X and another external time series D. It contains the AR(p) and MA(q) models of X and a linear combination of the last b terms of the time series D.

***Machine Learning-based techniques*:** These are two popular techniques used to carry out analysis of time series that can be considered part of the broader "machine learning" field:

Regression is a statistical method used to determine the polynomial function that is the closest to a set of points (in this case, the w values of the history window). *Linear regression* refers to the particular case of a polynomial of order 1. The objective is to find a polynomial such that the distance from each of the points to the polynomial curve is as small as possible and therefore fits the data the best. When the number of input variables is more than one, it is referred to as the Multiple Linear Regression. The Linear Regression equation is the same used in AR, but the weight estimation method differs.

Neural networks consist of an interconnected group of artificial neurons, arranged in several layers: an input layer with several input neurons; an output layer with one or more output neurons; and one or more hidden layers in between. For time series analysis, the input layer contains one neuron for each value in the history window, and one neuron for the predicted value in the output layer. During the training phase, it is fed with input vectors and random weights. Those weights will be adapted until the given input shows the desired output, at a learning rate ρ.

As previously stated, a group of time series analysis techniques try to identify the pattern that the series follows, and then use this pattern to extrapolate future values. Time series patterns can be described in terms of four classes of components: trend, seasonality, cyclical and randomness. The general trend (e.g. increasing or decreasing pattern), together with the seasonal variations that appear repeated over a specific period (e.g. day, week, month, or season), are the most common components in a time series. Input workloads of cloud applications may show different periodic components. The trend identifies the overall slope of the workload, whereas seasonality and cyclical determine the peaks at specific points of time in a short term and in a long term basis, respectively.

A wide diversity of methods can be used to find repetitive patterns in time series, including:

***Pattern matching*:** It searches for similar patterns in the history time series, that are similar to the present pattern. It is very close to the *string matching problem*, for which several efficient algorithms are available (e.g. Knuth-Morris-Prat [130]).

***Signal processing techniques*:** Fast Fourier Transform (FFT) is a technique that decomposes a signal time series into components of different frequencies. The dominant frequencies (if any) will correspond to the repeating pattern in the time series.

***Auto-correlation*:** In auto-correlation, the input time series is repeatedly shifted (up to half the total window length), and the correlation is calculated between the shifted time series and the original one. If the correlation is higher than a given threshold (e.g. 0.9) after s shifts, a repeating pattern is declared, with duration s steps.

A basic tool for time series representation is the *histogram*. It involves distributing the values of the time series into several equal-width bins, and representing the frequency for each bin. It has been used in the literature to represent the resource usage pattern or distribution, and then predict future values.

5.6.2 Review of Proposals

In the context of elastic applications, time series analysis have been applied mostly to predict workload or resource usage. A simple moving average could be used for this purpose, but with poor results [132]. For this reason, authors have applied MA only to remove noise from the time series [118], [119], or just to have a comparison yardstick. For example, Huang et al. [135] present a resource prediction model (for CPU and memory utilization) based on double exponential smoothing, and compare it with simple mean and weighted moving average (WMA). ES clearly

obtained better results, because it takes into account the history records w (not only the input window q) of the time series for the prediction. Mi et al. [136] also used Brown's double ES in order to forecast input workload of real traces (World Cup 98 and ClarkNet), and obtained good accuracy results, with a small amount of error (a mean relative error of 0.064 for the best case).

The auto-regression method has also been used for resource or workload forecasting ([48], [137], [132], [142], [94], [117]). For example, Roy et al. [117] applied AR for workload prediction, based on the last three observations. The predicted value is then used to estimate the response time. An optimization controller takes this response time as an input and computes the best resource allocation, taking into account the costs of SLA violations, leasing resources and reconfiguration. Kupferman et al. [48] applied auto-regression of order 1 to predict the request rate (requests per second) and found that its performance depends largely on several manager-defined parameters: the monitoring-interval length, the size of the history window and the size of the adaptation window. The history window determines the sensitivity of the algorithm to short-term versus long-term trends, while the size of the adaptation window determines how far into the future the model extends.

ARMA models are able to capture characteristics of a time series such as the input workload or the CPU usage. Fang et al. [134] found it useful to predict the future CPU usage of VMs. However, they remark the computational cost of this technique, that includes the choice of p and q, the estimation of the coefficients of each term and other parameters.

The history window values can also be the input for a neural network [138] [139] or a multiple linear regression equation [48], [122], [138]. The accuracy of both methods depends on the input window size. Indeed, Islam et al. [138] obtained better results when using more than one past value for prediction. Kupferman et al. [48] further investigated the topic and found that it is necessary to balance the size of each sample in the window, to avoid overreacting, but also to maintain a correct level of sensitivity to workload changes. They propose regressing over windows of different sizes, and then using the mean of all predictions. Another important issue is the choice of the prediction interval r. Islam et al. [138] propose using a 12-minute interval, because the setup time of VM instances in the cloud is typically around 5-15 min. In another context, Prodan and Nae [139] use a neural network to predict a game load (i.e. the number of entities or players) in the next two minutes. The neural network obtained better accuracy than moving average and simple exponential smoothing.

Most time series analysis techniques have been applied to vertical or horizontal scaling separately. Dutta et al. [140] claim that vertical scaling has limited range but has lower resource and configuration costs, while horizontal scaling can allow the application to achieve a much larger throughput but at a potentially higher

cost. For this reason, they combine both VM resizing in case of small increments in the request rate, and apply horizontal scaling for major changes in the input workload. They use a polynomial regression to estimate the expected number of requests for the next interval. Fang et al. [134] focus on vertical scaling (CPU and memory) for *regular* changes in workload, whereas horizontal scaling is applied in order to handle sudden spikes and flash crowds.

Time series forecasting (associated to proactive decision making) can be combined with reactive techniques. For example, Iqbal et al. [141] proposed a hybrid scaling technique that utilizes reactive rules for scaling up (based on CPU usage) and a regression-based approach for scaling down. After a fixed number of intervals in which response time is satisfied, they calculate the required number of application-tier and database-tier instances using polynomial regression (of degree two).

Some auto-scaling proposals use time series analysis techniques that deal with pattern identification, applied to the input workload [143], [131], [132], [133]. The most complete comparison of this class of techniques is done by Gong et al. [132]; they propose using FFT to identify repeating patterns in resource usage (CPU, memory, I/O and network), and compare it with auto-correlation, auto-regression and histogram. Pattern matching, proposed by Caron et al. [143] [131], has two main drawbacks: the large number of parameters in the algorithm (such as the maximum number of matches or the length of the predicted sequence), that highly affect the performance of the algorithm, and the time required to explore the past history trace.

Simple histograms have also been used by some authors to predict the resource usage of applications, considering the mean of the distribution [142], or the mean of the bin with the highest frequency [132].

Time series analysis techniques are very appealing for implementing auto-scalers, as they are able to predict future demands arriving to elastic applications. Having this information, it is possible to provide resources in advance and deal with the time required to start up new VMs or add resources to a particular instance. Despite the potential of this set of techniques, their main drawback relies on the prediction accuracy, that highly depends on the target application, input workload pattern and/or burstiness, the selected metric, the history window and prediction interval, as well as on the specific technique being used. Efforts should be focused on automating the selection of the best prediction technique for a particular application or application class.

Table 5.5 contains a summary of the articles reviewed in this section.

5.7 Summary and Conclusions

Cloud computing environments allow users to dynamically scale their applications. The key problem is how to lease the right amount of Cloud computing environments allow users to dynamically scale their applications, on a pay-as-you-go basis. Application re-dimensioning can be implemented effortlessly, adapting the resources assigned to the application to the incoming user demand. However, the identification of the right amount or resources to lease in order to meet the required Service Level Objectives, while keeping the overall cost low, is not an easy task. Many techniques have been proposed for automating this application scaling. We propose a classification of the techniques into five main categories: static threshold-based rules, control theory, reinforcement learning, queuing theory and time series analysis. Besides, we analyze the different experimental platforms used in the literature to test the auto-scaling techniques, including application benchmarks and workload generators.

Table 5.4: Summary of the reviewed literature about control theory techniques

Ref	Auto-scaling Techniques	H/V	R/P	Metric	Monitoring	SLA	Workloads	Experimental Platform
[118]	CT: PI controller	V	R	Job progress	Sensor library	Job deadline	Batch jobs	Custom testbed. HyperV + 5 applications (ADCIRC, OpenLB, WRF, BLAST and Montage)
[119]	CT: PI controller (Proportional thresholding) + Exponential Smoothing for performance variable	H	R	CPU load, request rate	Hyperic HQ (Xen)	-	Synthetic. Different number of threads.	Custom testbed. Xen + ORCA + simple web service
[120]	CT: PI controller (Proportional thresholding)	H	R	CPU load, request rate	Hyperic SIGAR. 10 seconds	-	Synthetic.	Custom testbed. Xen + Modified CloudStone (using Hadoop Distributed File System)
[121]	CT: MIMO adaptive controller + ARMA (performance model)	V	P	CPU usage, disk I/O, response time	Xen + custom tool. 20 seconds	Response time	Synthetic and realistic (generated with MediSyn)	Custom testbed. Xen + 3 applications (RUBiS, TPC-W, media server)
[115]	CT: Adaptive controllers + QT	H	R/P	Number of requests, service rate	Simulated	Number of requests not handled	Real. World Cup 98.	Custom simulator in Python
[114]	CT: Adaptive, Proportional controller + QT	H	R/P	Number of requests, requests in buffer, service rate	Simulated. 1 minute	-	Real. World Cup 98 and Google Cluster Data	Custom simulator in Python
[122]	CT: Gain-scheduler (adaptive) + Smoothing splines (performance model) + Linear Regression	H	P	Number of requests, number of servers, response time	20 seconds	Response time	Synthetic (Faban generator)	Real provider. Amazon EC2 + CloudStone benchmark
[123]	CT: Self-adaptive controller + Kriging model (performance model)	H	P	Number of incoming and enqueued requests, number of VMs	-	Execution time	Synthetic (Batch jobs)	Custom testbed. Private cloud + Sun Grid Engine (SGE)
[124]	CT: fuzzy controller	V	R	Number and mixture of requests, CPU load	Custom tool. 20 seconds	Reply rate	Real (World Cup 98) and Synthetic (Httperf)	Custom testbed. VMware ESX Server + Java Pet Store.
[125]	Fuzzy model	V	P	Number of queries, CPU load, disk I/O bandwidth	Xentop. 10 seconds	Response time, throughput	Synthetic and realistic (based on World Cup 98)	Custom testbed. Xen + 2 applications (RUBiS and TPC-H)
[126]	CT: Adaptive controller + Fuzzy model (+ ANN)	V	P	Number of requests, resource usage	Custom tool. 3 minutes	End-to-end delay	Synthetic. (Pareto distribution)	Simulation
[127]	CT: Adaptive SISO and MIMO controllers + Kalman filter	V	P	CPU load	Custom tool. 5-10 seconds	Response time	Synthetic (Browsing and bidding mix)	Custom testbed. Xen + RUBiS application

Table 5.5: Summary of the reviewed literature about techniques based on time series analysis

Ref	Auto-scaling Techniques	H/V	R/P	Metric	Monitoring	SLA	Workloads	Experimental Platform
[131]	TS: Pattern matching	H	P	Total number of CPUs	100 seconds	Number of serviced requests, cost	Real cloud workloads: from Animoto and 7 IBM cloud applications	Analytical models
[132]	TS: FFT and Discrete Markov Chains	V	P	CPU load	Libxenstat library. 1 minute	Response time	Real. World Cup 98 and ClarkNet. Also Synthetic trace	Custom testbed. Xen + RUBiS + part of Google Cluster Data trace for CPU usage.
[133]	TS: FFT and Discrete-time Markov Chain	V	P/R	CPU load, memory usage	Libxenstat library. 1 second	Response time, job progress	Synthetic. Based on World Cup 98 and EPA web server	Custom testbed. Xen + 3 applications (RUBiS, Hadoop MapReduce, IBM System S)
[134]	TS: ARMA	Both	P	Number of requests, CPU load	-	Prediction accuracy	Real traces. Collected from real applications and a real data center	Custom testbed. Xen and KVM
[135]	TS: Brown's double ES. Compared with WMA	-	P	CPU load, memory usage	Simulated	Prediction accuracy	Synthetic traffic generated with simulator	CloudSim simulator
[136]	TS: Brown's double ES.	-	P	Number of requests per VM	10 minutes	-	Real. World Cup 98 and ClarkNet. Synthetic. Poisson distribution	Custom testbed. TPC-W
[137]	TS: AR	H	P	Login rate, number of active connections, CPU load	Simulated	Service not available (login), energy consumption	Real. From Windows Live Messenger (login rate, number of active connections)	Simulator.
[94]	TS: AR + Threshold-based rules	Both	P	Number of requests	Zabbix	-	Synthetic	Hybrid: Amazon EC2 + Custom testbed (Xen + Eucalyptus + Php-Collab application)
[117]	TS: AR	H	P	Number of users in the system	-	Response time, VM cost, application reconfiguration cost	Real. World Cup 98	No experimentation on systems
[138]	TS: ML – Neural Network and (Multiple) LR + Sliding window	H	P	CPU load (aggregated value for all VMs)	Amazon CloudWatch. 1 minute	Prediction accuracy	Synthetic. TPC-W generator, constant growing	Real provider. Amazon EC2 and TPC-W application to generate the dataset
[139]	TS: ML – Neural Network (compared to MA, last value and simple ES)	H	P	Number of entities (players)	2 minutes	Prediction accuracy	Synthetic. Entities (players) with different behaviors	Simulator of a MMORPG game
[140]	TS: Polynomial regression	Both	P	Number of requests	Custom tool. 1 minute	Response time, cost for VM, application license and reconfiguration	Real traces. From a production data center of a company	Custom testbed. KVM + Olio
[141]	Threshold-based rules (scale out) + TS, polynomial regression (scale in)	H	R/P	CPU load (scale out), number of requests, number of VMs (scale in)	1 minute	Response time	Synthetic. Httperf	Custom testbed. Eucalyptus + RUBiS
[142]	TS: AR(1) and Histogram + QT	H	P	Request rate and service demand	Simulated. 1, 5, 10 and 20 minutes	Response time	Synthetic (Poisson distribution) and Real (World Cup 98)	Custom simulator + algorithms in Matlab

Chapter 6

Dynamic threshold rules for Auto-scaling

In the previous chapter we have described the state-of-art of auto-scaling techniques and proposed a taxonomy for all of them. The present chapter compares some representative techniques from each category in a simulated scenario and proposed a simple, yet effective variant of rules, with dynamic thresholds.

Our target scenario is a web-type application, deployed over a pool of homogeneous VMs. We will focus on the load balancer and the business tier. User requests arrive the application following a given pattern (in this case, based on a real workload). The load balancer will distribute requests among the VMs, based on different policies: random, round-robin or least-connection. Round-robin policy distributes requests in turns, whilst the least-connection one consists of assigning requests to the least loaded VM, i.e., the VM that is executing the least number of tasks.

Each task is assigned to a single VM. Request execution time is denoted as its *expected service time*, and in our simulated environment it is known *a-priori*. VMs adopt a time-sharing policy, so all incoming tasks will be accepted, but service time will increase accordingly to the number of tasks in the CPU. In this context, the *service time* is the lapse since a task is assigned to a VM, until the response is received back.

The auto-scaler will perform horizontal scaling actions: adding (scaling out) or removing VMs (scaling in). Many performance metrics could be applied to trigger auto-scaling actions. The most typical ones in the literature are CPU load, service time and input request rate, but many others have been proposed such as memory used by the VMs, or available bandwidth. Scaling decision may be

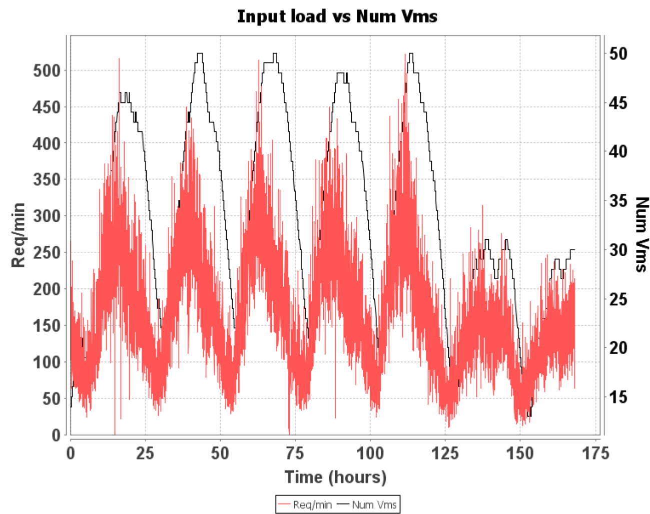


Figure 6.1: ClarkNet trace (the workload) vs. number of VMs assigned

taken based on a single metric or a combination of two or more. For simplicity, this study focuses on the use of a single metric.

The performance of each technique may be evaluated in terms of the total VM cost and the total number (or ratio) of SLO violations. Each cloud provider uses its own billing scheme. VM hours can be charged per minute or per hour (and in that case, partial hours are accounted as full hours). The cost may include not only VM running hours, but also other resource usage (e.g. disk or bandwith) and services (e.g. monitoring system).

Figure 6.1 shows an example of a real workload of one week duration. The number of VMs (represented with a thin line) varies according to the system load (number of requests per minute), from 10-15 VMs, up to 50 VMs at workload peaks.

6.1 Selected Auto-scaling Techniques

Many diverse auto-scaling techniques have been proposed in the literature. In previous chapter, a taxonomy consisting of five categories was defined: static threshold-based rules, time series analysis, control theory, reinforcement learning and queuing theory. We will compare techniques from 3 of these categories.

Static threshold-based rules are typically used by cloud providers such as Amazon EC2. A simple example: if $CPU > 70\%$, then scale out; if $CPU < 30\%$,

then scale in. It is quite difficult to set the correct thresholds and this must be done manually. An incorrect adjustment will cause oscillations in the number of VMs, and therefore, lead to bad performance. After each scaling decision, a cooldown period can be applied, during which no scaling will be performed. This cooldown period reduces the oscillations in the number of VMs and can be applied not only to rules, but to any auto-scaling technique. In RightScale's [85] variation of rules, each VM votes independently, based on rules like those explained before, whether to scale or not. Then, a simple democratic voting is performed to decide the scaling action.

Time series analysis includes a number of methods that use a past history window of a given performance metric in order to predict its future values (*proactive techniques*). In this case, we consider three methods: moving average, exponential smoothing and linear regression. Moving average calculates the mean of the n last values. Exponential smoothing assigns exponentially decreasing weight to each value in the time series. Last, linear regression tries to fit a linear equation to the last values (where x is the time and y is the performance metric value), and then, it estimates a future value.

Control theory has been applied to automate the management of different systems. The typical controller is the Proportional Integral Derivative (PID) controller. In this study, we have implement a simplified version called I (Integral) controller that obeys to the following equation:

$$u_k = u_{k-1} + K_i(y_k - y_{ref}) \tag{6.1}$$

where u_k is the new value for the manipulated variable (new number of VM); y_k is the performance metric value (CPU load), y_{ref} is the set point or target value; and K_i is the integral gain parameter. In other words, the integral controller will adjust the number of VMs in order to maintain the performance metric value (e.g. CPU load) as closer as possible to the target value.

Lastly, we have proposed a new auto-scaling technique, based on rules, that tries to overcome the limitation of using static thresholds. In this case, the upper and lower thresholds will be adjusted based on another set of rules. First, if SLO violations occurs for a certain time (e.g. 5 minutes), the threshold range will be *widened.* For example, a 60-40% configuration for upper-lower thresholds may be adapted to 80-20%. This makes the system less reactive to workload changes, the auto-scaler will be less eager to remove VMs, and thus, the number of SLO violations will be reduced. The opposite case is when no SLO violations have taken place during the last period of time, so the threshold range is *narrowed.* Then, the system becomes more reactive. The modification to use dynamic thresholds has also been applied to RightScale's voting system.

The dynamic threshold-based rules constitute a novel contribution of this work. All the remaining auto-scaling techniques discussed in this paper, and many

additional ones, are described in detail in [144].

Table 6.1: Performance values of load balancing policies with static threshold-based rules

Rules		R-Robin		Random		Least-con.	
UT	DT	Cost	SLOv	Cost	SLOv	Cost	SLOv
60	20	5756.70	0.15	5967.40	4.53	5727.40	0.13
70	30	4677.00	0.86	4799.50	10.36	4625.90	0.81
80	30	4330.10	1.46	4335.30	15.50	4294.40	1.39
80	40	4037.20	3.24	4053.40	21.21	4004.70	3.15
90	30	4026.70	2.66	3907.00	23.71	4002.50	2.35
90	10	5172.80	0.49	4855.80	15.46	5151.50	0.34

6.2 Experiments

This section presents the results for the different auto-scaling techniques and analyzes them. We have extended the functionality of the CloudSim cloud simulator to carry out the experiments. As described in Section 7.2.1, a web-type application is simulated, which is composed of a business tier (15 initial VMs), and a load balancer. All VMs are homogeneous: one core of 1GHz; other resources such as memory, disk or bandwith have not been considered.

Request execution time is generated based on a uniform distribution, that ranges from 3 to 7 seconds. The arrival time is taken from a real workload trace, the ClarkNet trace [145]. It contains 1654276 requests, that arrive in a clear cyclic pattern (see Figure 6.1): daytime has more workload than the night, and the workload on weekends is lower than that taking place on weekdays.

CPU load (mean load of all VMs) is used as the performance metric. The monitoring interval of one minute and scaling decisions are taken based on the performance metric value from the last minute. Scaling actions add or remove a single VM (other options could be considered, such as adding/removing a percentage of the current number of running VMs). Two different values of boot-up time (time to effectively put to work a new VM) are considered: 0 and 10 minutes. After a scaling decision, a cooldown period is applied. This cooldown period is equal to the boot-up time, plus an extra period of 5 minutes (that is necessary to see the effect of an scaling decision on the system). When a scale-in decision is made, the chosen VM is not removed until all tasks that are being serviced on it finish. Therefore, in a subsequent scale-out action, instead of

creating a new VM, we can use one of these VMs that are pending to be removed and avoid the boot-up time.

Each experiment is repeated 10 times. Auto-scaling techniques are compared attending to the mean VM cost (*VMcost*) and the mean number of SLO violations (*SLOv*). In this scenario, the Amazon EC2 [8] billing scheme is applied: VM hours are charged per running hour, and partial hours are accounted as full hours. Boot up time is not charged. An SLO violation occurs when the service time of a task is greater than or equal to $5 * expectedtime$.

Results are divided into several subsections. Focusing on the rules, we analyze the impact of applying different load balancing policies, and also compare static vs. dynamic thresholds. Then, several reactive and proactive techniques are compared.

6.2.1 Impact of load balancing policy

Three load balancing policies have been tested: round-robin, random and least-connection. For the comparison, we will focus on the results for static threshold-based rules, and instant boot up of VMs. Results are gathered in Table 6.1.

Random policy shows the worst results, up to 23.71% of SLO violations. Its nature may cause an unbalanced distribution of tasks among the VMs. Round-robin and least-connection policies both show similar results, but the latter performs a little better. The performance of these two policies will depend on the homogeneity in the duration of the tasks. If all tasks have the same execution time, round robin will be the best option. Otherwise, if tasks have different execution times, least-connection policy is able to distribute the load more evenly among the VMs. For the rest of the experiments, we will use round robin as the load balancing policy.

6.2.2 Reactive techniques

Reactive techniques considered in this paper include rules, RightScale's voting system, two proposals based on dynamic thresholds and an integral controller. Results for two different values of boot-up time (0 and 10 minutes) are shown in Table 6.2.

Static vs dynamic threshold-based rules

In this section we compare typical rules based on static thresholds (*Rules*), also combined with a voting system (*RightScale*), against our proposal of adapting thresholds depending on the number of SLO violations. Results are shown for the round-robin load balancer policy, 0 minute boot up time and CPU load as the performance metric (see Table 6.2).

Dynamic thresholds consider 50% as the mid range, 90% as the upper limit and 10% as the lower limit. These limits have been established in order to avoid a 100-0 and/or 95-5% threshold-configurations, that lead to a situation where few scaling actions are performed and the number of VM remains mainly constant. Variations of thresholds are of 5%. Cooldown time is applied for scaling rules, but not for threshold adaptation (this is checked every minute).

Regarding both simple rules and RightScale's voting system based on static thresholds, the performance depends highly on the selection of threshold values. Some configurations result in low VM cost, but in consequence, the number of SLO violations is excessive. Despite the fact that the lowest number of SLO violations is obtained using certain values of static thresholds, our proposal based on dynamic values is able to maintain an acceptable SLO compliance (0.50% for simple rules and 0.58-0.83% for the RightScale's approach), regardless the initial threshold values. Combining dynamic thresholds with simple rules seems to be the best option (rather than using a voting system), since the number of SLO violations remains nearly constant.

Integral controller

The integral controller is different from the rest of reactive auto-scaling techniques, because it considers a target CPU value, instead of trying to keep CPU within an acceptable range (delimited by upper and lower thresholds). It directly calculates the required number of VMs (this value is rounded up). K value has been selected using a trial and error method. This parameter is really difficult to set, and a bad choice causes high oscillations in the number of VMs. Although it is the technique that achieves the lowest number of SLO violations (0.04%), it does so at a high cost and, therefore, we can not consider integral controller as the best choice.

As a general conclusion, reactive techniques cannot anticipate to changes and a boot-up time of 10 minutes causes an increase in both the VM cost and the number of SLO violations. The performance of each auto-scaling technique highly depends on parameter selection, which is difficult to be done manually. Besides, a bad configuration causes a high number of SLO violations and dissatisfied clients. Our proposal based on dynamic thresholds is able to improve the general performance, by adjusting the thresholds automatically, without human intervention.

6.2.3 Proactive techniques

Three proactive methods (based on time series analysis) are considered: moving average (MA), linear regression (LR) and exponential smoothing (ES). They predict the mean CPU load for the next period (1 minute ahead) and this value is used with threshold-based rules to decide the next scaling action. Among the threshold configurations for the scaling rules, we have selected the three

that obtained the lowest number of SLO violations in the reactive case: 60-20%, 70-30% and 90-10%. Table 6.3 contains the results for two boot-up times: 0 and 10 minutes.

Again, results highly depend on parameter configuration, history window and smoothing factor. In case of MA, the best result is obtained for a history window W equal to 2; higher values of this parameter lead to an increase in the number of SLO violations. On the contrary, for LR case, it is better to use a larger history window ($W = 10$). Last, ES is applied with different values of the smoothing factor α. The lowest number of SLO violations is obtained with $\alpha = 0.6$ (in two out of three cases), which balances the weight given to new values (0.6), and the weight assigned to the past values.

MA and ES techniques are less dependent on the parameter values than LR. The latter can vary from 9.70% of SLO violations with a $W = 2$ to 2.91% with $W = 10$ (with 70-30% thresholds and 10 minutes of boot-up time).

Looking at the number of SLO violations for both instant and 10-minute boot-up times, the lowest results are obtained by ES with $\alpha = 6$ (0.13/0.23%), nearly followed by MA with $W = 2$ (0.15/0.24%) and last, LR with $W = 10$ (0.15/0.26%). All three techniques improve the results of static threshold-based rules (0.15/0.33% for 60-20% threshold values).

6.2.4 Comparing Reactive vs. Proactive techniques

Figure 6.2 shows a global perspective of auto-scaling techniques. Regarding proactive techniques, only results for 60-20% threshold configuration are presented, as they obtained the lowest number of SLO violations.

In general, the performance worsens when the boot-up time of VMs is 10 minutes in terms of SLO violations. In contrast, the overall cost is quite similar for the two values of boot-up time. In almost all cases, for both reactive and proactive approaches, the number of SLO violations highly depends on the parameter configuration. The lowest number of SLO violations is achieved by the integral controller and RightScale's voting system, but both require a fine tuning of two or more parameters. However, proactive techniques (MA and ES), combined with 60-20% threshold-based rules, are less dependent on parameters, and still improve the performance of simple rules. Finally, proactive techniques have been tested with rules for simplicity, but they may be combined with any method such as an integral controller or RightScale's voting system.

6.3 Summary and Conclusions

Cloud computing environments have a great potential for running scalable applications, thanks to their elasticity nature. Auto-scalers are the key to the efficient

use of elastic resources, because they enable the user to adjust resources to needs, while complying with SLO and reducing the cost. Using simulation, we have tested some representative auto-scaling techniques, comparing them in terms of overall cost and SLO violations. We have also proposed a method based on rules with dynamic thresholds, that improves the performance of simple, static threshold-based rules. The main conclusion extracted is that any auto-scaling method is very dependent on the parameter tuning.

Table 6.2: Results for reactive techniques, based on CPU load, for two different values of boot up time. UT and DT refer to the Up and Down thresholds, respectively.

Technique	Parameters			0 min		10 min	
				VMcost	SLOv	VMcost	SLOv
Rules	UT: 60	DT: 20		5756.70	0.15	5575.80	0.33
	UT: 70	DT: 30		4677.00	0.86	4402.60	2.91
	UT: 80	DT: 30		4330.10	1.46	4155.80	4.92
	UT: 80	DT: 40		4037.20	3.24	3764.10	8.89
	UT: 90	DT: 30		4026.70	2.66	3914.90	7.13
	UT: 90	DT: 10		5172.80	0.49	5136.20	0.79
Dynamic thresholds	UT: 60	DT: 20		5175.50	0.50	5127.60	0.87
	UT: 70	DT: 30		5169.50	0.50	5127.50	0.87
	UT: 80	DT: 30		5168.90	0.51	5127.50	0.87
	UT: 80	DT: 40		5168.30	0.51	5126.80	0.90
	UT: 90	DT: 30		5168.70	0.51	5127.40	0.88
	UT: 90	DT: 10		5173.50	0.48	5136.30	0.79
RighstScale	UT: 60	DT: 20	V 75%	5813.70	0.14	5741.20	0.23
	UT: 60	DT: 20	V 95%	5712.00	0.11	5665.70	0.19
	UT: 70	DT: 30	V 75%	4656.90	0.78	4467.00	2.63
	UT: 70	DT: 30	V 95%	4631.10	0.84	4523.00	2.27
	UT: 80	DT: 40	V 75%	4001.70	3.12	3771.00	8.54
	UT: 80	DT: 40	V 95%	3940.20	3.14	3777.80	8.76
	UT: 90	DT: 10	V 75%	5134.60	0.61	5089.40	1.10
	UT: 90	DT: 10	V 95%	5090.40	0.87	5092.50	1.34
Dynamic thr. RightScale	UT: 60	DT: 20	V 75%	5138.30	0.58	5094.40	1.04
	UT: 60	DT: 20	V 95%	5095.00	0.78	5105.10	1.22
	UT: 70	DT: 30	V 75%	5136.50	0.58	5094.40	1.04
	UT: 70	DT: 30	V 95%	5087.90	0.81	5105.10	1.22
	UT: 80	DT: 40	V 75%	5114.90	0.60	5092.90	1.05
	UT: 80	DT: 40	V 95%	5086.00	0.82	5105.10	1.22
	UT: 90	DT: 10	V 75%	5131.50	0.59	5079.40	1.05
	UT: 90	DT: 10	V 95%	5085.30	0.83	5105.10	1.22
IController	K: -0.01	Target: 60%		6474.60	0.04	6538.50	0.29
	K: -0.01	Target: 65%		5658.00	0.16	5492.60	0.71
	K: -0.01	Target: 70%		5089.20	0.44	4906.00	1.81
	K: -0.01	Target: 75%		4602.30	1.17	4467.20	4.07
	K: -0.01	Target: 80%		4207.70	2.40	4098.40	7.55

Table 6.3: Results for proactive techniques, based on CPU load, for two different values of boot up time

T.	UT LT	Par.	0 min		10 min	
			VMcost	SLOv	VMcost	SLOv
MA	60 20	W:2	5699.70	0.15	5638.80	0.24
		W:3	5659.80	0.16	5600.90	0.31
		W:5	5572.30	0.18	5551.20	0.36
		W:10	5479.00	0.23	5393.70	0.41
	70 30	W:2	4585.80	0.78	4414.00	2.59
		W:3	4535.70	0.90	4423.20	2.73
		W:5	4539.30	1.07	4502.20	2.27
		W:10	4505.20	1.43	4469.80	2.62
	90 10	W:2	5063.00	0.88	5090.90	1.27
		W:3	5008.70	1.25	5066.20	1.57
		W:5	5079.30	1.57	5050.70	2.13
		W:10	4945.20	2.37	5003.40	3.41
ES	60 20	α:.1	5364.50	0.30	5371.20	0.45
		α:.3	5544.50	0.18	5548.20	0.33
		α:.6	5751.00	0.13	5715.10	0.23
		α:.9	5735.70	0.15	5603.00	0.29
	70 30	α:.1	4453.30	1.85	4428.30	3.14
		α:.3	4565.70	1.00	4514.40	2.26
		α:.6	4596.20	0.71	4474.00	2.22
		α:.9	4662.70	0.78	4436.80	2.72
	90 10	α:.1	4801.40	3.74	4844.60	4.65
		α:.3	5092.10	1.68	5144.60	2.32
		α:.6	4997.70	1.02	5039.30	1.40
		α:.9	5099.00	0.58	5090.80	0.90
LR	60 20	W:2	5602.30	0.71	4993.90	2.23
		W:3	5594.70	0.45	5057.50	1.76
		W:5	5604.70	0.26	5345.70	0.53
		W:10	5605.60	0.15	5471.60	0.26
	70 30	W:2	4666.90	2.69	4144.70	9.70
		W:3	4691.00	1.88	4183.30	8.06
		W:5	4695.50	1.20	4261.50	5.87
		W:10	4525.30	0.97	4380.30	2.91
	90 10	W:2	4734.60	1.42	4293.70	6.23
		W:3	4836.50	1.09	4581.60	3.52
		W:5	4901.80	0.73	4794.60	1.64
		W:10	4994.70	0.98	4944.00	1.51

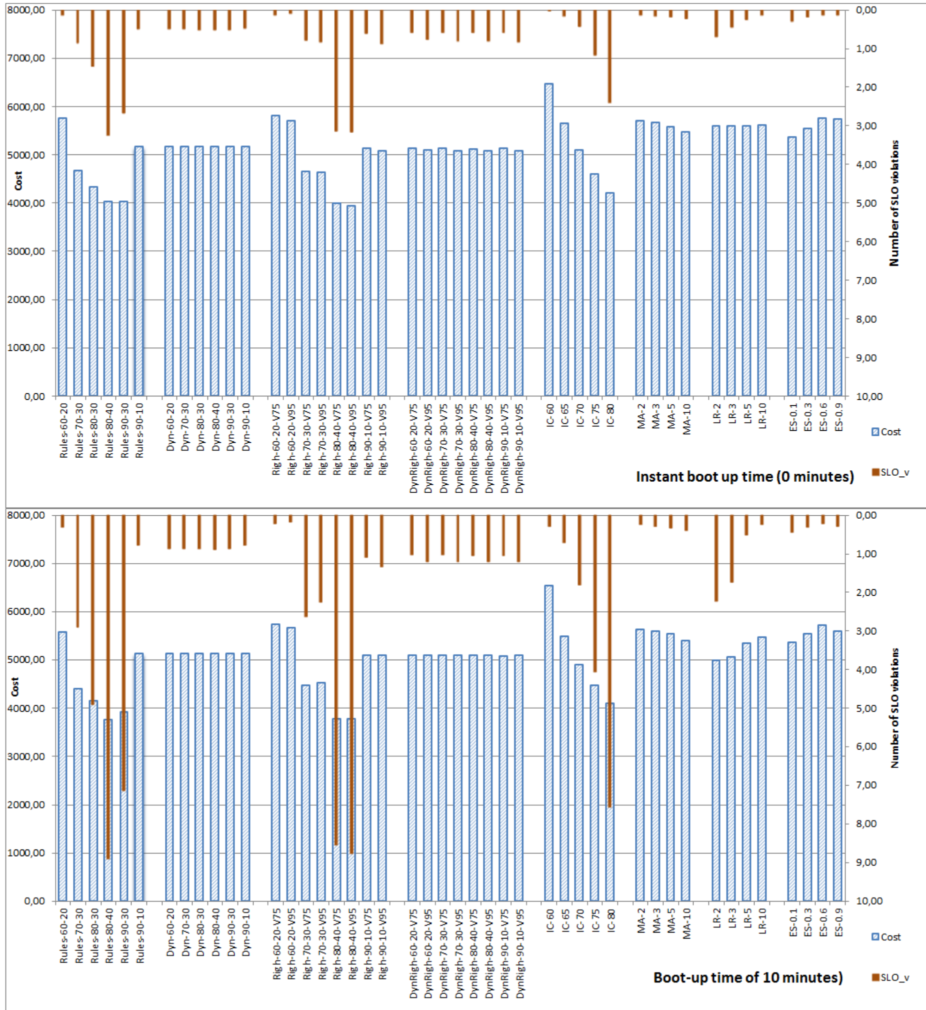


Figure 6.2: VM cost and number of SLO violations for auto-scaling techniques, with instant boot-up time (top) and 10 minutes of boot-up time (bottom).

Chapter 7

An online algorithm to detect the Noisy Neighbor Problem

Cloud data centers are shared among many applications. Resource sharing leads to interferences among VMs or containers, the so-called noisy neighbor effect. This can lead to serious performance issues and ultimately affect the quality of service in cloud applications. The current chapter proposes an unsupervised algorithm to detect anomalies in the performance and tests in a real environment using different types of applications.

7.1 Our requirements

Given the literature, we have come up with the following requirements for our algorithm:

- *Application-agnostic:* That is, the algorithm should no require any a-priori knowledge of the application. We have seen that each application has different resource usage patterns (check Figure 2.2).

- *With unknown normal state distributions*: Even when these are known, they are subject to change given VM changes (up/down) or variations in input workload pattern.

- *Online and lightweight*: The delay cannot be huge and the computation must allow frequent calls to the algorithm.

- *Correlation of metrics*: We want a solution that captures anomaly situations that reflect as a correlation between different metrics (e.g. CPU and memory).

As stated by EBATS [**?**], statistical methods offer a promising solution for anomaly detection and this is why we chose this approach.

This paper proposes an unsupervised online algorithm for detecting anomalies in VMs or containers resource usage profile. By definition, *normal* behavior will be more frequent than the *abnormal* behavior. Our approach is to model this behavior for different time periods, and compare them. Models that are really different can be considered as anomalous.

Our proposed algorithm includes the following specific contributions: (1) a way of modeling the unknown behavior of an application, represented by multiple resources, (2) that deals with the dynamic nature of the application through retraining, and (3) able to identify anomalies without any a-priori knowledge.

The resource usage of an application can be modeled as a Gaussian Mixture Model (GMM). This method has the capacity to fairly represent many types of load distributions as a combination of several normal distributions, with their respective means and variances [71]. Models are constantly retrained to capture the changes in the normal behavior of the application. A significant change in the current state in contrast to frequent *normal* behavior is interpreted as an anomaly [146].

7.2 Anomaly Detection Algorithm

This section provides a top-down description of the proposed anomaly detection algorithm. The target scenario and the system architecture are described first. Then, a general overview of the algorithm is provided, followed by the techniques that match the requirements of our scenario. Finally, we provide the specific pseudo-code of the algorithm.

7.2.1 Scenario and System Architecture

The target scenario is a physical server with a virtualization layer that enables resource sharing among different services or applications. Each service runs on a separate VM. Techniques such as overbooking or consolidation, as well as workload changes are factors that may lead to VM interference. Our goal is to detect anomalies in the resource usage of a particular VM, produced by VM interference, the so-called noisy neighbor effect [55].

The system architecture is depicted in Figure 7.1. The input to the system is a set of time series, related to resource usage metrics (e.g. CPU load, memory

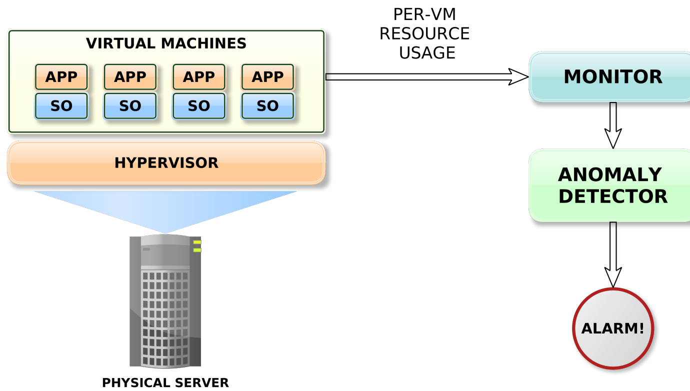


Figure 7.1: General architecture for anomaly detection system

consumption), and optionally, application-specific performance indicators (e.g. response time). The monitoring module collects the input data at the given granularity. The anomaly detection module contains the core logic of the algorithm. It receives the resource usage from the last period as input data. Whenever an anomaly is detected, it raises an alert. These alarms can be published back to a monitoring tool.

7.2.2 Challenges

There are many challenges to overcome for anomaly identification:

- The normal behavior is unknown and application-specific.
- Several anomaly classes may appear, with different statistical properties.
- The normal behavior of a system evolves with time, due to external factors: workload change, availability of resources.

Next section describes the general strategy to address each of these challenges.

7.2.3 General Strategy

The obvious approach to identify an anomaly is to compare it with the past normal behavior of the application. However, we are assuming that there is no previous historical info about the statistical properties of the normal behavior or the anomalies. Furthermore, this normal behavior of the application evolves with

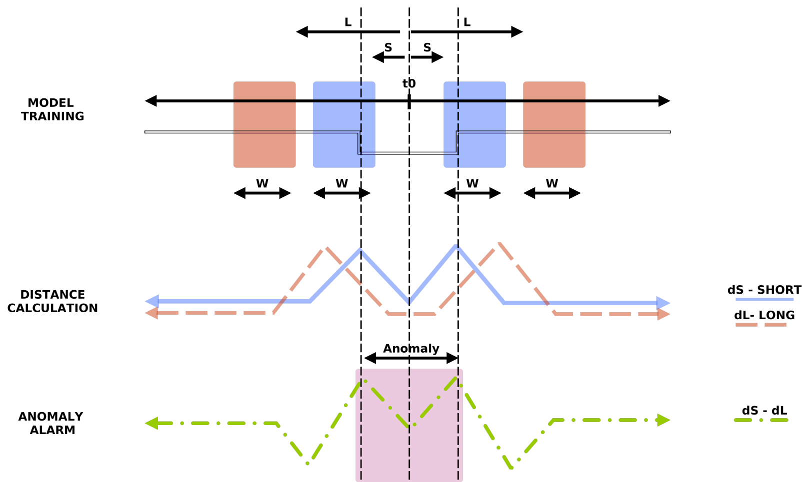


Figure 7.2: General strategy for anomaly detection algorithm. Models are trained based on windows of size W samples. By comparing models located at a short S lag with respect to t_0, short distance line (in *blue* color) is obtained. Similarly, long distance line (in *red*) is obtained by comparing models located at a longest lag L. The result of calculating the difference between short and long distance values is depicted as a *green* line. The anomaly appears between the two consecutive *peaks* (shaded in *red*).

time. This motivates the use of an *unsupervised* approach for anomaly detection, together with continuous state retraining, based on a sliding-window technique.

Anomalies are rare events (in comparison to normal data) and represent abrupt changes in the normal behavior. Intuitively, we can identify a potential anomaly when our current model (state) is quite different from the recent past, but still quite similar to the longest past.

Our general strategy is depicted in Figure 7.2. We want to determine if our current point in time t_0 is anomalous or not. For this purpose, we look at both past and future data with respect to t_0. More specifically, we check for the state at a short lag S from t_0 and at a long lag L (in both directions). If a change is detected in the short-term while the long-term states remain stable, we have found an anomaly.

The application behavior over a certain period, that is, the resource usage during this time is defined by a *model*. We use W samples to train such model. We also need a *distance metric* that scores the similarity between two models.

By calculating the difference between the short-term and long-term distances, we obtain the green line in Figure 7.2. The last step is to detect such pattern ($min - max - min - max - min$). The anomaly appears between the two consecutive local max.

7.2.4 Techniques

Dirichlet Process Gaussian Mixture Model

Clustering is the task of grouping data points in such a way that the data points in the same group (denoted as *cluster*) are more similar to each other. In the case of a GMM (Gaussian Mixture Model), the process is similar to learning a probability distribution per cluster. Each cluster (group), also denoted as a *component*, is modeled as a Gaussian distribution $\mathcal{N}(x : \mu, \Sigma)$ (mean μ and covariance matrix Σ), and its associated weight *pi*. In this manuscript, full covariance matrix is used to train GMMs, in order to capture relationships (correlations) among attributes. In other scenarios, a diagonal covariance matrix would simplify the computational cost.

The probability distribution function (PDF) of a GMM with K Gaussian components can be expressed as follows:

$$P(x) = \sum_{i=1}^{K} \pi_i \mathcal{N}(x : \mu_i, \Sigma_i)$$

where x is an ordered set of features in the model. In our problem, it represents the different variables that describe the VM or container resource usage (e.g. CPU load, memory consumption).

In contrast to a GMM, a Dirichlet Process Gaussian Mixture Model (DPGMM) does not assume any a-priori number of components, i.e., it can be understood as an *infinite* GMM [147]. This characteristic is extremely appealing for scenarios in which there is no intuition about the nature of the data. The DPGMM training algorithm selected for this manuscript is called Variational Inference for Dirichlet Process Mixtures [148].

Measuring Similarity between DPGMMs

There are many distances in the literature to measure the similarity between different clusterings (i.e. the resulting set of clusters after applying a clustering algorithm on a dataset). For this specific context, we will focus on similarity metrics for comparing probability distributions and adaptations for GMMs (applicable also to DPGMMs).

A GMM model is defined by a set of components. Each component can be expressed in terms of its mean value, μ, for each attribute; the covariance matrix, Σ; and the weight associated to that component, π. Let us define two GMM models P and Q. P is composed of a set N components, $P = \{P_1, P_2, \ldots, P_N\}$; similarly, Q is composed of M components, $Q = \{Q_1, Q_2, \ldots, Q_M\}$. Components can be defined as $P_i = \mathcal{N}(x : \mu, \Sigma)$ and $Q_j = \mathcal{N}(x' : \mu', \Sigma')$.

Our goal is to define a distance $D(P, Q)$ that scores the similarity between two GMMs P and Q. We can find many distances d in the literature to compare Gaussian distributions, that is, they can be used to compare components. We can leverage some of these distances d and find a general form to compare GMMs based on its components. Given a distance d between every pair of components $P_i \in P$ and $Q_j \in Q$, the following D approach can be used to calculate the distance between P and Q GMMs Amara et al. [149]:

$$D(P, Q) := \sum_{i=1}^{N} \sum_{j=1}^{M} \pi_i \pi_j d(P_i, Q_j)$$

Two of the most popular distances for probability distribution comparison are Kullblack Liebler divergence [150] and Bhattachariyya distance [151, 152]. Additionally, we have selected C2, [153], and proposed a simple, naive distance that only considers the mean μ of each component, called mean-value distance.

Each of the distances is described separately:

Kullblack Liebler divergence or relative entropy: Let P and Q be two PDF, the Kullback–Leibler divergence of Q from P is defined as:

$$d_{KL}(P||Q) = \int_{-\infty}^{\infty} P(x) \ln \frac{P(x)}{Q(x)} dx$$

Kullback-Leibler divergence is not symmetric. In contrast, Jensen-Shannon divergence, or simply called Symmetric Kullback-Liebler (SKL), is a symmetrized version of the Kullback–Leibler divergence:

$$d_{SKL}(P, Q) := 1/2 * (d_{KL}(P\|Q) + d_{KL}(Q\|P))$$

The closed formula to calculate the SKL distance between two multi-variate Gaussian distributions (thus, two GMM components P_i, Q_j) is given by:

$$d_{SKL}(P_i, Q_j) = \frac{1}{2}\left((\mu - \mu')^T(\Sigma^{-1} + \Sigma'^{-1})(\mu - \mu')\right) + \frac{1}{2}\left(tr(\Sigma^{-1}\Sigma') + tr(\Sigma'^{-1}\Sigma) - tr(2I)\right)$$

where μ and Σ are the mean and covariance matrix for P_i component; μ' and Σ' are the mean and covariance matrix for component Q_j; tr is the trace and I is the identity matrix.

The Kullback-Liebler-based distance between two GMMs is computed as the sum of the SKL distance d_{SKL} between every pair of components:

$$D_{KL}(P,Q) := \sum_{i=1}^{N}\sum_{j=1}^{M}\pi_i\pi_j d_{SKL}(P_i,Q_j)$$

Bhattacharyya-based distance : For two general probability distributions P and Q, Bhattacharyya distance is defined as follows:

$$d_{BH}(P,Q) = -\ln\int\sqrt{P(x)Q(x)dx}$$

In case that both distributions are gaussian (e.g. components P_i and Q_j), Bhattacharyya distance can be computed as:

$$d_{BH}(P_i,Q_j) = \frac{1}{8}(\mu-\mu')^T\left(\frac{\Sigma+\Sigma'}{2}\right)^{-1}(\mu-\mu') + \frac{1}{2}\ln\left(\frac{|\frac{\Sigma+\Sigma'}{2}|}{\sqrt{|\Sigma|\,|\Sigma'|}}\right)$$

where μ and Σ are the mean and covariance matrix for P_i component; μ' and Σ' are the mean and covariance matrix for component Q_j.

Similarly to Kullback-Liebler divergence, we can adapt the d_{BH} distance to GMMs as follows:

$$D_{BH}(P,Q) := \sum_{i=1}^{N}\sum_{j=1}^{M}\pi_i\pi_j d_{BH}(P_i,Q_j)$$

C2 distance: Given two general probability distributions P and Q, $C2$ distance is defined as follows:

$$C2(P,Q) = -log\left[\frac{2\int P(x)Q(x)dx}{\int[P(x)]^2dx + \int[Q(x)]^2dx}\right]$$

When P and Q are GMMs, $C2$ distance can be calculated as follows:

$$C2 := -\log\left[\frac{2\sum_{i,j}\left\{\pi_i\pi'_j\sqrt{\frac{|V_{ij}|}{e^{k_{i,j}}|\Sigma_i||\Sigma'_j|}}\right\}}{\sum_{i,j}\left\{\pi_i\pi_j\sqrt{\frac{|V_{ij}|}{e^{k_{i,j}}|\Sigma_i||\Sigma_j|}}\right\} + \sum_{i,j}\left\{\pi'_i\pi'_j\sqrt{\frac{|V_{ij}|}{e^{k_{i,j}}|\Sigma'_i||\Sigma'_j|}}\right\}}\right]$$

where

$$V_{ij} = (\Sigma_i{}^{-1} + \Sigma'_j{}^{-1})^{-1}$$

$$k_{i,j} = \mu_i^T \Sigma_i{}^{-1}(\mu_i^T - \mu_j'^T) + \mu_j'^T \Sigma_j'{}^{-1}(\mu_j'^T - \mu_i^T)$$

Note that μ_i, Σ_i and μ_j, Σ_j refer to components in GMM P, while μ_i', Σ_i' and μ_j', Σ_j' refer to components in GMM Q.

Mean-value distance: This distance is just considered as a naive case, for comparison purposes. Given two general probability distributions P and Q, the distance $D_{CC}(P, Q)$ can be calculated based their means. In case that P and Q are GMM, it can be computed as follows:

$$D_{CC}(P, Q) := \frac{\sum_{i=1}^{N} \min_{j=1,\dots,M} d(\mu_i, \mu_j') + \sum_{j=1}^{M} \min_{i=1,\dots,N} d(\mu_i, \mu_j')}{(N+M) \max\limits_{\substack{i=1,\dots,N \\ j=1,\dots,M}} d'(\mu_i, \mu_j')}$$

where $d(\mu_i, \mu_j')$ is the euclidean distance between the means of μ_i and μ_j' of the components P_i and Q_j, respectively.

7.2.5 Selected Techniques

We need to select the concrete model, distance metric and pattern-finding technique to fully implement our algorithm.

The input data is a set of time series that describe the resource usage of a particular VM (or container) over a certain period of time. This data follows an unknown multivariate probability distribution. It is known that any multivariate density can be approximated with a mixture of normals, given enough components [154]. Indeed, Yu et al. [72] used a GMM (Gaussian Mixture Model) to represent the normal machine performance. For a d dimensional dataset, this would be k multivariate normal distributions: $x \sim \mathcal{N}(\mu, \Sigma)$.

An important parameter in mixture modeling is the number of mixture components k. With too few components, the trained model may not be able to approximate the true underlying model effectively; in contrast, with too many components, it may overfit the data and increase the computational complexity. Alternatively to GMMs, Dirichlet Process Gaussian Mixture Models (DPGMM) do not require any a-priori number of components. DPGMM is our choice to represent the resource usage profile of an application, as the number of required components is unknown and application-specific. DPGMMs are further described in 7.2.4.

We need a way to score the dis-similarity between any pair of models. There exist several metrics in the literature which quantify the dis-similarity between Gaussian distributions. We can adapt these metrics to provide a single score of the difference between two DPGMM models. These are the distance metrics selected

for our current implementation: Kullback-Leibler divergence [150], Bhattacharyya distance [151, 152], C2 distance [153] and the naive mean-value based distance. Formulas can be found in 7.2.4.

Finally, the anomaly is reflected as a concrete pattern in the short minus long distance time series. Let us denote d_S as the distance values from comparing short-term models over time; and similarly, d_L to designate distance values between long-term models. In Figure 7.2, d_S and d_L are represented in blue and red colors, respectively. We need to find a concrete pattern, $min - max - min - max - min$, in the resulting time series calculated as $d_S - d_L$ (see green line in Figure 7.2).

For the current implementation, we propose the following *ad-hoc* pattern-detection approach that works efficiently in our experiments. The Anomaly Area A is equal to $A = (W + L) \cdot 2$, that is, it covers the training data used for short and long term models. First, we raise pre-alarms when $d_S - d_L$ is greater than a given percentile value $p = 0.75$. This threshold is used to ignore non-significant variations in $d_S - dL$ time series. Then, for each positive pre-alarm, we look for at least 25% prealarm values in the next $A = (W + L) \cdot 2$ period. In that case, we raise an anomaly alarm for the whole period A, that starts with the first pre-alarm.

7.2.6 Algorithm Description

The general pseudo-code for the proposed anomaly detection method can be found in Algorithm 3. Each major step is described separately below:

```
Data: Input: Resource usage
Result: Anomaly alert
while true do
    Read new window of data (size ws) ;
    Data pre-processing (moving average) ;
    Train DPGMM model for current period t;
    Calculate distances at diffent lags ;
    Check for alarm;
end
```

Algorithm 3: General algorithm for Anomaly detection

From data to model The input data is composed of several time series representing the resource usage of a particular VM or container (e.g. CPU load, memory consumption). Raw data is quite noisy, so a pre-processing step is applied to each time series, consisting of a moving average filter. We assume that the data is normalized, between 0 and 1. A DPGMM is trained at each timestamp based on recent data. The number of samples in the training window is denoted as W.

Table 7.1: List of parameters related to the anomaly detection algorithm

Type	Notation	Name	Definition	Value
DPGMM	α	Dispersion	Alpha parameter corresponding to DPGMM model	0.01
	k_{max}	Max components	Maximum number of components considered by DPGMM learning algorithm	5
	n_{iter}	Number of iterations	Number of iterations for the DPGMM training algorithm	25
Detection	W	Training window	Number of samples used to create a new DPGMM model at step t, using period $[t - W/2, t + W/2]$.	120-240
	S	Short-term lag	Comparing models at $t_0 - S - W/2$ and $t_0 + S + W/2$	0-180
	L	Long-term lag	Comparing models at $t_0 - L - W/2$ and $t_0 + L + W/2$	60-180
	as	Anomaly size	Anomaly duration in number of samples	10-60
	A	Anomaly area		$2 \cdot (WS + L)$

From models to distance scores Every pair of DPGMM is compared to obtain a similarity score. There are specific distance metrics that are suitable to compare DPGMM models: Kullback-Leibler divergence [150], Bhattacharyya distance [151, 152], C2 distance [153] and the naive mean-value based distance. For each time t_0, we calculate the values for short S and long L lags. The short lag implies comparing models at $t_0 - S - W/2$ and $t_0 + S + W/2$; similarly, for the long lag we use models at $t_0 - L - W/2$ and $t_0 + L + W/2$.

From distance scores to alerts Intuitively, the anomaly alarm should be raised between the two local, consecutive maximums found in the difference between long and short distance values. As described in previous section, an alert is raised for each pre-alarm detected by the pattern-finding approach.

7.2.7 Parameters and constraints

All parameters are summarized in Table 7.1. DPGMM models have the advantage of not having to specify the a-priori number of clusters, only the maximum number of components k_{max}. In our case, this parameter can be set to a highly enough value that might cover any case of anomaly. We need to tune the dispersion parameter α and the number of iterations n_{iter} needed for the learning algorithm to converge. We will use the full covariance version that is more computationally demanding than other approaches (spherical, diagonal or tied), but far more precise. We need to carefully tune the parameters related to the model creation algorithm and alarm system: training window W, the short-term S and long-term L lags.

By definition, the detection delay is equal to $W + L$, always assuming that $S > 0$, $L > 0$ and $S < L$. Besides, the anomaly size as should be at most half of

the window training size, thus: $as < W/2$.

7.3 Experimental Framework

7.3.1 Evaluation

We consider a successful detection (True Positive, TP) if the algorithm raises at least one alarm during the anomaly duration. Otherwise it will be considered a False Negative, FN. We will also account for False Positives, FP, or alarms raised when there is no anomaly. The goal of our algorithm is to maximize the number of anomalies detected (True Positive) and to reduce the number of False Positives.

We derive the following metrics for evaluation. To assess the True Positive rate we will use Precision $P = \frac{TP}{TP+FP} = \frac{\text{\# of successful detections}}{\text{\# of total alarms}}$. We will use Recall to check for the False Alarm Rate, defined as $R = \frac{TP}{TP+FN} = \frac{\text{\# of successful detections}}{\text{\# of total anomalies}}$. Finally, the f_1 score provides a unique value between precision and recall $f_1 = 2 \cdot \frac{P \cdot R}{P+R}$.

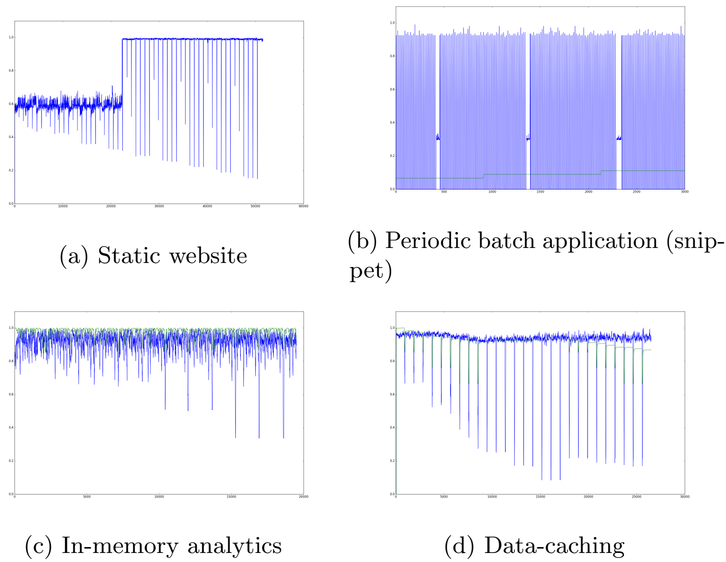
7.3.2 Application Benchmarks

We focus our experimentation on CPU and Memory anomalies. We have selected a diverse set of benchmarks to target the different application types typically run on a cloud data center: a static web-page, a generic batch application, an in-memory data analytic application, and a data caching server. The workload patterns are depicted in Figure 7.3.

Static website: This application is an instance of the RUBiS benchmark[1] that processes requests for the home page. The goal is to perform preliminary experiments in which the only resource affected is the CPU; it uses a pretty constant amount of memory. For the client side, we use the httpmon[2] tool to generate incoming requests to the application.

Periodic batch application: A special-case application that tries to solve a fixed number of sudokus. If there are not enough resources, the remaining sudokus will be enqueued for the next time period. In contrast to the previous application, the normal behavior of the application is periodic, meaning that the CPU load shows a repetitive pattern (see Figure 7.3b).

[1] `http://rubis.ow2.org/` *[Last accessed 17-July-2017]*
[2] `https://github.com/cloud-control/httpmon` *[Last accessed 17-July-2017]*



(a) Static website

(b) Periodic batch application (snippet)

(c) In-memory analytics

(d) Data-caching

Figure 7.3: Workloads used in the experiments

In-Memory Analytics: It is part of the standard cloud benchmarking framework called CloudSuite [155]. This application uses Apache Spark and runs a collaborative filtering algorithm (named *alternating least squares*, provided by Spark MLlib). The input dataset contains user-movie ratings and it is stored in memory.

Data-caching: This benchmark is also part of CloudSuite framework. This benchmark uses the Memcached data caching server, simulating the behavior of a Twitter caching server using a Twitter dataset.

Static web-page serves well as a naive application to test the algorithm accuracy, due to a stable CPU load and near-constant memory consumption. In contrast, Sudoku application shows a repetitive normal behavior that increases the complexity of anomaly detection. Spark applications run on different stages with variable CPU and memory requirements, that might be challenging to differentiate from real anomalies. Finally, Memcached server is mainly targeted to create memory anomalies.

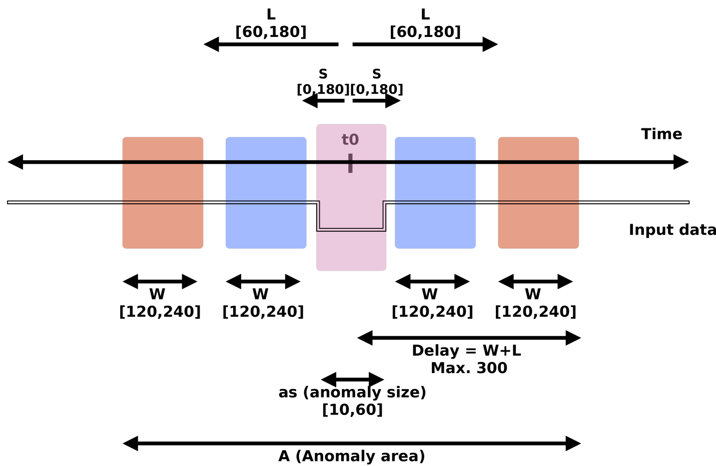


Figure 7.4: Sample representation of main parameters related with anomaly detection algorithm and the values considered in the experiments. The input data is a dummy two-valued time series, which represents a constant normal behavior, and a short anomaly.

7.3.3 Testbed

Selected applications are single-layered and are run as Docker containers, on top of a VM. (Static webpage requires a workload generator that is run on a separate VM). *Cgroups* tool can be used to restrict the resource cap assigned to the container and thus, simulate anomalies derived from the noisy-neighbor effect. A Python script interacts with Docker and uses *cgroups* to modify the CPU and memory limits for certain periods of time, according to an input file (called action file). Another Python-based script collects the metrics from CPU and memory usage stats reported by *cgroups*. CPU is defined as the sum of *User* and *System*, while memory usage is the *memory usage*. Experiments are carried out on a 4-core (Intel(R) Xeon(R) CPU E31240 at 3.30GHz) desktop computer with 16 GB of memory.

7.3.4 Parameter Selection

Figure 7.4 depicts the relationship among all the parameters and respective value ranges considered for the current experimental setup. We assume that anomalies have a certain duration in time, defined as anomaly size as. For the current experiments, anomaly durations are set between 10 and 60 samples. Given that the training window size W must be at least twice the maximum anomaly, 120

samples (2·60) is set to be the minimum value for W. The algorithm by definition has a $W + L$ delay. We set the maximum detection delay allowed to 5 minutes (300 samples, in seconds). To satisfy all constraints, W must take values within the range [120,240]. The remaining $(300 - W)$ correspond to the values set for L lag. Given the L value, we select the short lag S between 0 and L value.

DPGMMs require two main parameters: maximum number of components and dispersion parameters. We have found empirically that setting $max_k = 5$ and $\alpha = 0.01$ work well in our scenarios. The DPGMM learning algorithm converges after 25 iterations (n_{iter}) for the training sizes considered in the experiments.

7.4 Analysis of Results

This section has the objective of answering the following questions: (1) is our algorithm feasible in computational terms, (2) which is the best parameter configuration for each scenario, and (3) what is the performance of the detection algorithm under different workloads, (4) how does our algorithm perform in comparison with other baseline methods.

7.4.1 Computational requirements of our proposed algorithm

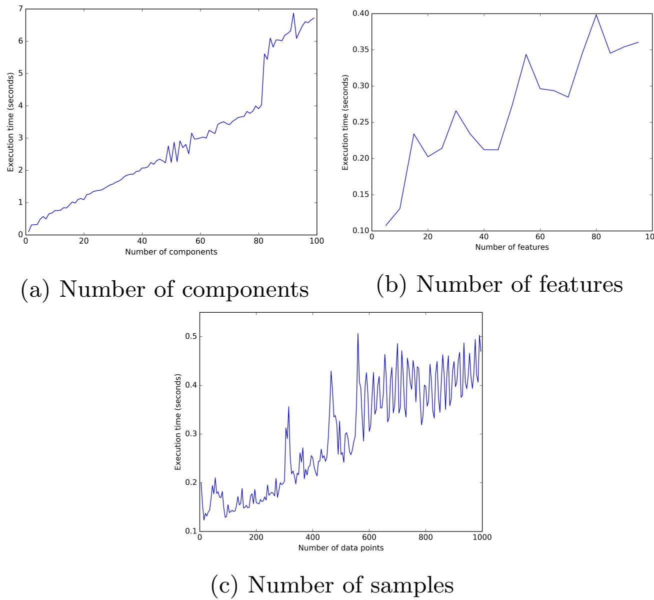
Our algorithm consists on 3 main stages. The most costly stages in our proposed anomaly detection algorithm are DPGMM training and distance metric calculation. We devote a separate section to each of them. We explore the theoretical computational complexity and look at the empirical execution times. Finally, we explain our experience about CPU and memory requirements of the whole algorithm.

DPGMM training time

The DPGMM training algorithm selected for this manuscript is called Variational Inference for Dirichlet Process Mixtures [148]. Its theoretical computational complexity is as follows:

The complexity of this implementation is linear in the number of mixture components K and data points (samples) X. With regards to the dimensionality (number of features, n), it depends on the type of covariance used. In this particular implementation, that uses *full* covariance matrix, the complexity is equal to $O(K \cdot X \cdot n^2 + K \cdot n^3)$ (the cubic term arises from the need to invert the covariance/precision matrices and compute its determinant).

Most anomaly detection methods consider metrics in isolation, simplifying the computational cost. In contrast, we prefer to consider all possible correlations



(a) Number of components

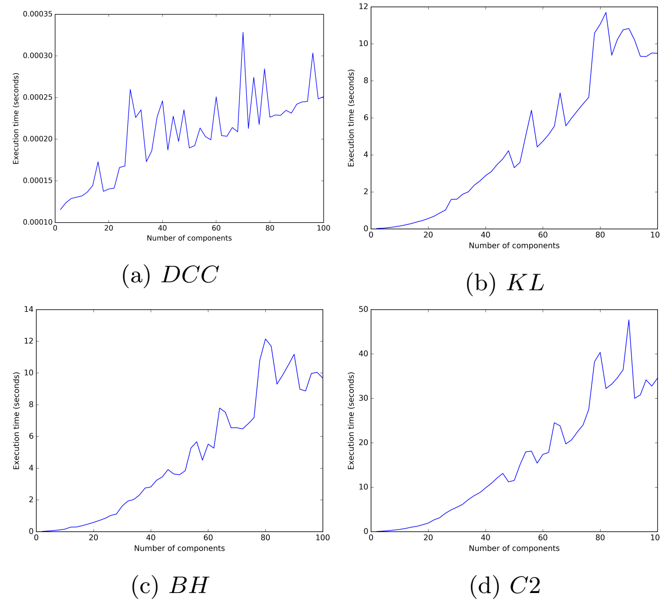
(b) Number of features

(c) Number of samples

Figure 7.5: Model training times for different number of components, features and samples

among metrics, as there might be anomalies reflected as a combination of changes in several metrics simultaneously (e.g. CPU and memory). Thus, we choose the full covariance matrix. Alternatively, a diagonal or spherical covariance matrix would simplify calculations: the former assumes that each component has its own diagonal covariance matrix, while the latter assumes that each component only has a single variance. With any of these two configurations, the training algorithm cost becomes linear: $O(K \cdot N \cdot n)$, given that $X >> n$ and $X >> K$.

In order to better illustrate the behavior of the training algorithm, we have performed some empirical testing in terms of the number of features, components and samples. Figure 7.5 shows the average execution time based on each variable. Each plot shows the training times of varying one variable (number of features, components and samples) and keeping the other two variables constant. Each execution time is the average of 5 iterations. It can be seen in the Figure that the training time grows linearly with the number of samples, features and components. The training times required for any number of samples (up to 1000 data points) and any number of features (up to 100) are almost negligible: training times are below one second. The number of components plays a bigger role, bringing the training time up to 10 seconds with 100 components. Still, this model training



(a) DCC (b) KL

(c) BH (d) $C2$

Figure 7.6: Distance calculation times for different number of components

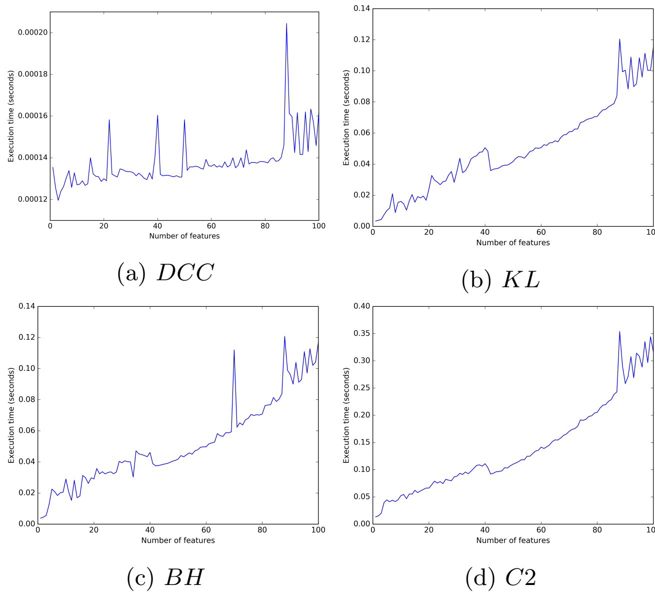
time is feasible. In our experiments, we use up to 5 components and the algorithm successfully captures the anomalies in the applications.

Distance metric calculation time

Let us define two GMM models P and Q. P is composed of a set N components, $P = \{P_1, P_2, \ldots, P_N\}$; similarly, Q is composed of M components, $Q = \{Q_1, Q_2, \ldots, Q_M\}$. Our proposed distance metrics (which are mean-value distance DCC, Kullblack Liebler divergence KL, Bhattachariyya distance BH and $C2$ distance) calculate the sum of distances between each pair of components in models P and Q. Thus, the base computational complexity for distance calculation $O(N \cdot M)$. Additionally, for $C2$ distance metric, it is $O(max(N, M)^2)$.

Here we analyze the empirical execution times of our own implementation of each distance metric, for different $N + M$ total number of components, and n number of features, Figures 7.6 and 7.7, respectively. Note that distance metric calculation is independent of the number of samples, as it only uses the mean and covariance matrix of each component.

We can observe how all distances calculation increases linearly with number of components and number of features. As expected, DCC distance metric has really



(a) *DCC* (b) *KL*

(c) *BH* (d) *C*2

Figure 7.7: Distance calculation times for different number of features

low computational requirements (in the order of milliseconds), as it only operates on component means. *BH* and *KL* distance metrics perform quite similarly, with up to 3 seconds for 100 features, and around 30 seconds for 100 components. Finally, *C*2 distance metric is the most costly in computational terms, requiring 1.5 minutes for up to 100 components, and 9 seconds for 100 features. All distance calculation seems computationally feasible: in our experiments, we will be using 2 features and up to 5 components. However, some initial conclusions can be made for distance preference: *KL* and *BH* distance metrics offer a good trade-off while operating on means and covariance matrices (thus, taking into account metric correlations); *DCC* distance metric could be a good choice for certain scenarios where very low execution times are mandatory.

CPU and memory requirements

Overall, our algorithm at each step requires one model training and two distance metric computations computations (short and long term). Results show that model training can be done in 10 seconds, and both distance metric calculations in around 5 seconds (choosing any of the metrics). The cost of identifying the anomaly pattern is negligible.

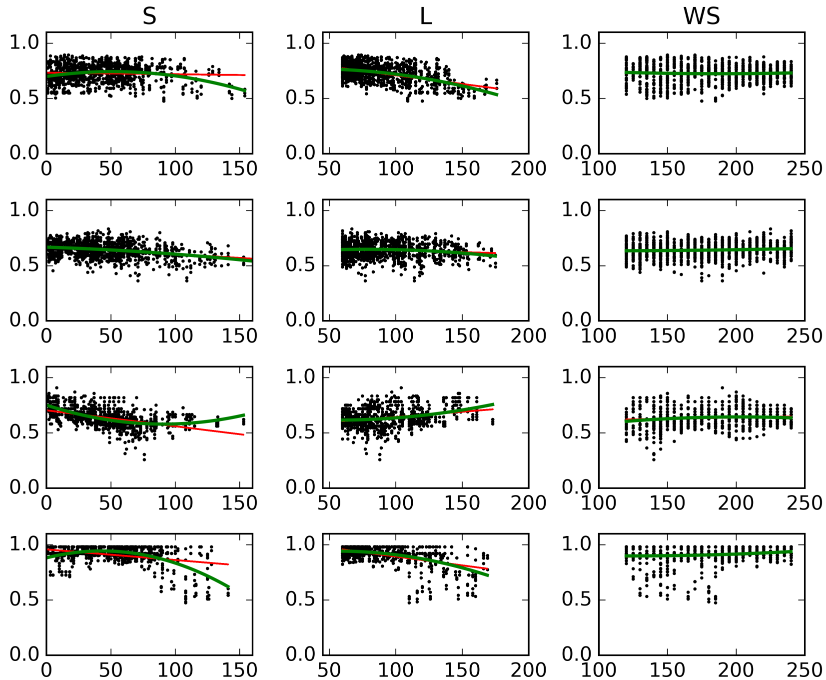


Figure 7.8: Plot for parameter S, L and W (x-axis), versus f_1 score (y-axis). Each row corresponds to one of the workloads: website, batch application, in-memory and data caching. Linear line and second-order regression are included, red and green lines respectively

The CPU impact of our algorithm highly depends on its execution frequency, but it would last a few seconds at most. We have been able to execute our algorithm with memory requirements around 150MB. Model storage does not play a big role, as we only need to store the means and covariance matrix for each component. That is, given each covariance matrix is a function of $O(n^2)$, the memory requirements to store a DPGMM is equal to $O(K \cdot n \cdot n)$, where n is the number of features and K is the number of components in the model.

The computational requirements of the algorithm can be easily adaptable by reducing the granularity of the samples. For example, a 5-second sampling could be enough, and it reduces the execution time by a factor of 5. Similarly, the memory requirements can be adjusted by adjusting the long-term lag, and thus, reducing the number of models stored.

7.4.2 Parameter Tuning

The anomaly detection algorithm depends on 3 main parameters: training window WS, short-term lag S, and long-term lag L. Figure 7.8 shows the f_1 score obtained for different W, S and L configurations for all workloads. Based on the results, we can recommend smaller S values, within the range [1-60], and L values in the range [60,80]. These configuration leads to higher f_1 values in all four workloads. In contrast, it is harder to extract conclusions about recommended training window sizes W.

Anomaly detection delay is given by the selected window size W and long-lag L (it is equal to $W + L$). Conveniently, the algorithm obtains higher f_1 scores for smaller L values [60,80], thus reducing the delay. With the same motivation, we would suggest using smaller training window sizes, always having in mind that W must be at least twice the size of the maximum anomaly duration.

Additionally, we evaluate different distance metrics: mean-based distance DCC, Kullback-Leibler-based KL, Bhattachariyya-based BH and $C2$ distance. Figure 7.9 shows a boxplot of f_1 score obtained for each distance measure under each workload. We cannot conclude about a best distance for all scenarios, because they perform quite similarly.

Instead, we can use the computational cost as selection criteria for distance metrics. Based on our current implementation, and taking DCC as the reference execution time x, KL takes $35x$, BH takes $71x$, and $C2$ takes $207x$. Obviously, DCC is the fastest distance, as it is calculated based solely on mean component values of the DPGMMs, ignoring the covariance matrix. Thus, it does not account for the relationships among input variables, and we do not encourage its use in real scenarios. In terms of computational cost and also good f_1 scores, we would suggest selecting the Kullback-Leibler-based distance.

7.4.3 Algorithm Evaluation

We have generated around 1000 configurations per each workload (four in total), randomly selecting values for all the parameters (distance measure, window size W, short-lag S and long lag L). Four distance measures have been considered: Kullback-Leibler KL, Bhattachariyya BH, $C2$ distance and mean-based distance DCC. Default configuration can be found in Table 7.1.

First, we evaluate the algorithm in terms of the True and False Positives. Let us look at the actual quality metrics: recall, precision and f_1-score. Figure 7.10 shows the boxplot for each of these metrics under the four different workloads. Higher precision values are achieved for data caching and static website workloads. In contrast, precision is much lower for Sudoku and In-memory workloads. Sudoku suffers from a significant number of false positives. In-memory workload is quite bursty in nature, so anomalies are harder to identify.

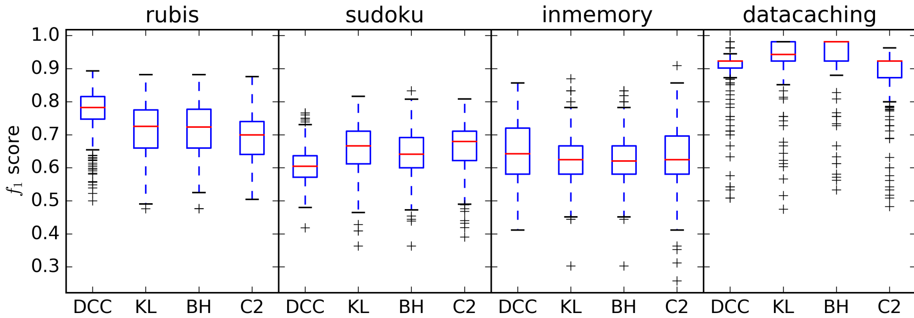
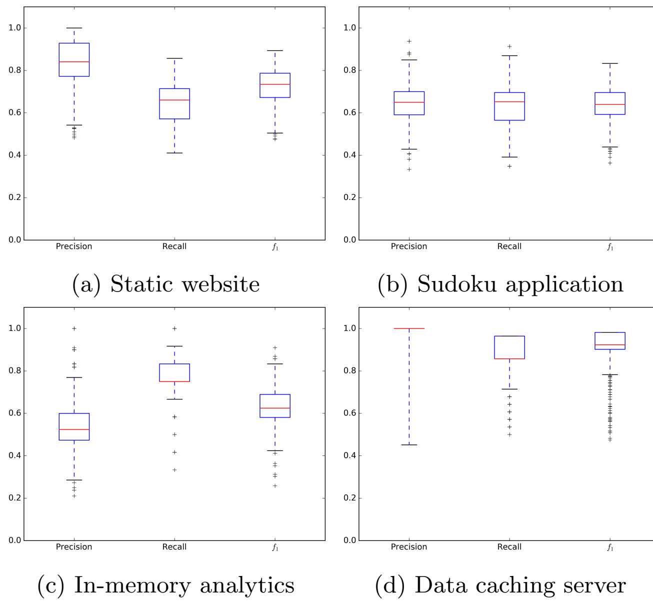


Figure 7.9: Boxplot of f_1 scores by workload and distance metric

Table 7.2: Summary of best f_1 score obtained for each Distance and workload. It gathers the parameter configuration used and the performance of the anomaly detection algorithm

Application	# of alarms	Parameters				Metrics				
		Distance	W	S	L	TP	FP	Precision	Recall	f_1
Website	56	DCC	170	17	74	46	1	0.98	0.82	0.89
		KL	150	47	67	45	1	0.98	0.80	0.88
		BH	150	47	67	45	1	0.98	0.80	0.88
		$C2$	120	74	89	46	3	0.94	0.82	0.88
Batch	23	DCC	185	75	106	18	6	0.75	0.78	0.77
		KL	240	40	60	20	6	0.77	0.87	0.82
		BH	225	48	67	20	5	0.80	0.87	0.83
		$C2$	210	49	80	19	5	0.79	0.83	0.81
In-memory	12	DCC	150	2	147	9	0	1.00	0.75	0.86
		KL	200	22	97	10	1	0.91	0.83	0.87
		BH	200	6	99	10	2	0.83	0.83	0.83
		$C2$	190	8	104	10	0	1.00	0.83	0.91
Data-caching	28	DCC	170	44	60	27	0	1.00	0.96	0.98
		KL	230	36	64	27	0	1.00	0.96	0.98
		BH	240	41	60	27	0	1.00	0.96	0.98
		$C2$	130	37	61	26	0	1.00	0.93	0.96

Table 7.2 presents the best result in terms of f_1 score, for each workload and distance. The best distance depends on the scenario. With the right configuration, our algorithm gets from 0.89 up to 0.98 in f_1 score. This translates into 82%-96% of anomaly detection percentage (that is, recall R). False alarm percentage ranges from 0 to 25% (equal to 1-precision P), depending on the scenario.



(a) Static website (b) Sudoku application

(c) In-memory analytics (d) Data caching server

Figure 7.10: Boxplot for Precision, Recall, f_1 score metrics, for each workload. It includes results for every parameter configuration: S, L, W and distance

7.4.4 Comparison against other methods

A common baseline method for outlier detection consists of modeling the metric distribution and considering as outliers the 2σ or 3σ values Zhang et al. [65]. For these experiments, we will consider both variants, 2σ and 3σ. However, given that the distribution probability is unknown, a simple Gaussian distribution is assumed, and trained based on the last window, or all the past history:

- Sliding window: The training window has been fixed to the 150 previous values.
- All History: It is trained with all past samples till the current time.

Table 7.3 presents the results of baseline methods for every application dataset. For the sake of comparison, we have included the results of our proposed algorithm, executed with the KL distance metric.

In general, we can observe that Sliding window approach fails most of the time to detect anomalies, as only using recent past is not a good representation of the normal behavior of an applications.

The rest of the evaluation focuses on All History method. Best results are achieved for data-caching workload with really high precision and recall: almost all anomalies are detected, and very few false positive alarms are raised. These results are similar to the ones achieved by our proposed algorithm. Both website and in-memory workloads exhibit good precision results, but there is an unbearable number of True positive alarms. This reflects on low precision values of 0.70/0.68 and 0.24 for website and in-memory, respectively. That is, a value of 0.24 means that from all alarms raised by All History baseline method, less than 1/4 are valid alarms. In contrast, our algorithm raises a really low number of false alarms (only 1 in both workloads), and achieves a precision of 0.98 for website workload, and 0.91 for in-memory workload.

Finally, the batch workload reveals the potential of our approach: All-history method fails to capture the normally periodic behavior of the application, and is not able to detect any of the anomalies (only 2 for 2σ case). The periodic changes in CPU consumption, though normal, are correctly interpreted as anomalies. Our algorithm successfully detects 87% of the anomalies in batch workload.

7.4.5 Discussion of results

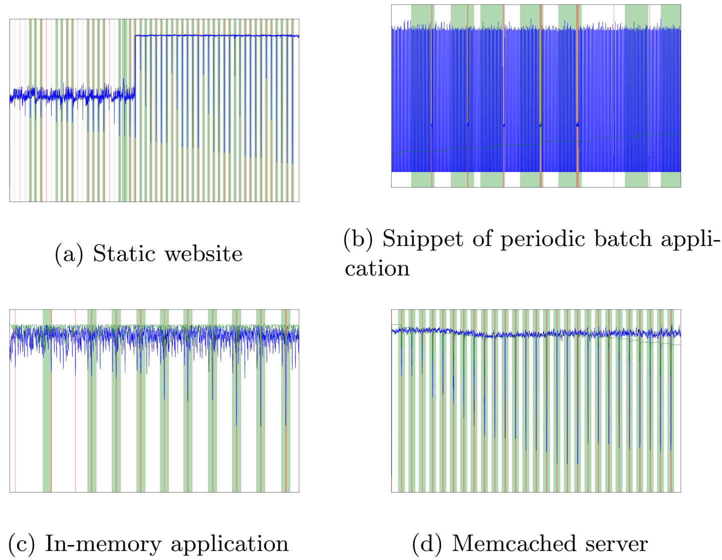
We have evaluated the performance of the anomaly detection algorithm in terms of some objective metrics: precision, recall and f_1 score. Let us analyze the reasoning behind that results. The performance of the algorithm seems to be affected by the general characteristics of the workload (resource usage) and also some properties of the anomaly itself.

Table 7.3: Results for baseline anomaly detection methods

Application	# of alarms	Method	Metrics				
			TP	FP	Precision	Recall	f_1
Website	56	Our algorithm, KL	45	1	0.98	0.80	0.88
		Sliding Window 2σ	54	638	0.08	0.96	0.14
		Sliding Window 3σ	51	114	0.31	0.91	0.46
		All History 2σ	39	17	0.70	0.70	0.70
		All History 3σ	17	8	0.68	0.30	0.42
Batch	23	Our algorithm, KL	20	6	0.77	0.87	0.82
		Sliding Window 2σ	2	1544	0.00	0.09	0.00
		Sliding Window 3σ	0	0	0.00	0.00	0.00
		All History 2σ	2	1552	0.00	0.09	0.00
		All History 3σ	0	0	0.00	0.00	0.00
In-memory	12	Our algorithm, KL	10	1	0.91	0.83	0.87
		Sliding Window 2σ	12	147	0.08	1.00	0.14
		Sliding Window 3σ	11	69	0.14	0.92	0.24
		All History 2σ	12	39	0.24	1.00	0.38
		All History 3σ	11	13	0.46	0.92	0.61
Data-caching	28	Our algorithm, KL	27	0	1.00	0.96	0.98
		Sliding Window 2σ	27	330	0.08	0.96	0.14
		Sliding Window 3σ	27	70	0.28	0.96	0.43
		All History 2σ	26	1	0.96	0.93	0.95
		All History 3σ	24	0	1.00	0.86	0.92

Anomaly properties An anomaly can be defined in terms of its *duration* and *cap*. The duration has already been denoted as as or anomaly size, while the *cap* can be understood as how severe the interference is. A slight interference would combine small cap and short duration, and it is typically harder to detect. For example, we can notice a few anomalies that are not detected for the static website workload (see Figure 7.11a). However, slight interference can be even harmless in terms of application performance, and thus, it seems more relevant to detect severe anomalies (these with higher duration and cap). Our algorithm works well under these conditions; in particular, the number of successful detections is higher for static website and memcached server workloads, as in both cases the cap is significantly higher.

Normal behavior We can identify three main factors in the workload characteristics that might affect anomaly identification: periodicity, stages, and burstiness. Each workload is representative of these ones: batch application for repetitive normal behavior; in-memory application for its different phases of execution; and the first phase of static website workload for its higher burstiness. *Data caching* workload presents ideal characteristics for anomaly detection: pretty smooth resource consumption, with visible anomalies, that are clearly reflected



(a) Static website

(b) Snippet of periodic batch application

(c) In-memory application

(d) Memcached server

Figure 7.11: Sample alarm results. Line colors: CPU load (blue), memory load (green); Real alarms (red), anomaly alarm (shaded in green)

on both CPU and memory consumption. In contrast, *in-memory* applications naturally present stages with variable memory/CPU consumption that can be easily interpreted as false anomalies. Same happens with the periodic behavior of the batch application. In both cases, recall (successfully detected anomalies) is lower, while the true positive rate is higher. Burstiness is another major factor in anomaly detection. Spikes in resource consumption might be misleading and cause false positive alarms. High burstiness hinders the identification of real anomalies (specially slight interference) and also increases the rate of false positives. The burstiness of the workload is addressed in our algorithm by applying a smoothing filter (e.g. moving average). This reduces the number of false positives, but affects negatively the detection of slight interferences, as the cap is smoothed.

In general, our algorithm seems to successfully detect a major number of interference anomalies for applications with different resource usage patterns. We have compared it against some baseline methods and demonstrated that our approach is able to cope with the unknown normal behavior of the application. For example, in the case of the Batch application, we are able to detect the periodic component in the resource usage profile as consider it as a non-anomaly. The general algorithm efficiency is dependent on the parameter tuning, but our analysis shows that the precision and recall stays in a reasonable range for all

the tested configurations. After some empirical analysis of the computational requirements, we can conclude that our algorithm is lightweight and thus, it can be used online for anomaly detection. Finally, let us note that the unsupervised approach of the algorithm is a huge key to its potential. The number of false alarms could be reduced by incorporating some problem-specific information. For example, by incorporating a rule-based detection that checks for response time or other indicator to filter False Positives, before actually triggering an alarm.

7.4.6 Advantages and disadvantages of our proposal

In all, we have proposed an online and lightweight solution for detecting anomalies caused by the noisy neighbor effect. As many other proposals in the literature, it is based on statistical methods, but it has some advantages. First, it is not assuming any previous history from the application and it is not specific to a particular application. Besides, the use of DPGMMs allows us to fairly approximate *any* resource usage profile, regardless of the probability distribution it might follow. It is even able to capture varying normal behaviors, like a repetitive pattern (see our Batch application). DPGMMs also accounts for metric correlations, which other algorithms in the state-of-art fail to capture. Finally, we cope with the application's changing nature by doing constant retraining.

Of course, all this advantages do not come for free. Our proposal does not need any previous information about the anomaly, as it follows an unsupervised approach. Logically, it is more prone to miss anomalies. Another drawback might be the computational requirements in case of using a high number of metrics. In this regard, we thought about using aggregated metrics: CPU load, memory, disk read/writes, etc. It is pretty easy to tune the algorithm to be really lightweight. Finally, our algorithms requires some parameter tuning to obtain best results, but experiments show that random configurations lead to reasonably good accuracy values and low rate of false positives.

Chapter 8

Spatially-Aware Distances for Hard Clusterings

In the previous chapter, we describe an anomaly detection algorithm with uses a soft clustering model (GMM) and proposes several distance metrics to measure the difference between different models. Inspired on this work, we also developed set of new distances focused on hard clusterings. The present chapter changes its focus from the data center to a more pure technical aspect. We will describe the set of proposed distances and the experiments carried out to confirm their validity.

8.1 Introduction

Clustering is the task of finding homogeneous groups within a set of data points. Let as denote *clustering solution* (also known as *clustering* or *partition*) as the resulting set of clusters, in which each data point is assigned to a *single* cluster. This is known as *hard clustering* and it will be the focus throughout the article.

Many clustering tasks require the comparison of two partitions. The most straightforward one is cluster evaluation, i.e., comparing a clustering solution against the ground-truth, or solutions created by different clustering algorithms. Other applications in the literature are consensus (or ensemble) clustering [156], which aims to combine several clustering solutions to produce a high quality one; or data stream

A metric should be applicable to *any* two clusterings without restrictions. Wagner and Wagner [157] mention three limitations that should be avoided:

cluster sizes, number of clusters, and dependencies between the clusterings (i.e. using the same data set). Metrics proposed in the literature usually impose big restrictions. Most traditional ones usually examine the number of points shared between two different clusterings [158, 159]. This imposes a big restriction: the comparison is limited to clusterings built upon the same data set. ADCO [160] measure was designed for the stream data clustering, and is applicable to compare clusterings with only partially shared data points. However, it assumes that both solutions contain the same number of clusters.

A distance measure should be able to capture the dis-similarity between any two different clustering solutions. Traditional measures for cluster evaluation only take into account the data item labeling, i.e. the cluster assignment [159, 161, 162]. It would be an appealing property if a distance measure can take into account the *spatial layout* of the data points. This includes both the *shape* and the *position* of the clusters from each partition.

We can intuitively affirm that there is no optimal measure for every case. Furthermore, there is no standard methodology for distance measure validation. Horta and Campello [163] identify a set of four properties that are appealing from a practical perspective: Maximum (identify equivalent clusterings), Discriminant (able to detect the best solution), contrast (provide better evaluations for progressively better solutions), and Baseline (have a predetermined expected value for randomly generated solutions). We also consider the computational cost as a relevant property for any measure.

The contributions of this article are: (1) a set of four metrics that do not impose any restrictions on the clusterings (except that attributes must lie in the same attribute space) and that take into account the spatial layout of the partitions, (2) validation of metric properties, both formally and empirically.

8.2 Background and Related Work

There is a great corpus of distance metrics in the literature, with different nature and ultimate goal (hard clustering[163], community detection in Graph clustering [164], subspace clustering [165]) or topics modeling [166]. Given their diversity, it seems difficult to propose a taxonomy that fits all of them. Several attempts can be found in the literature. For our current article, we will adopt the taxonomy presented in [164], that divides metrics into three key areas: set comparison, spatially-aware clustering cmparison and subspace clustering comparison. Subspace clustering falls out of the scope of this manuscript, as our target are hard clusterings that use the same set of attributes. Therefore, this section analyzes the state-of-art of the first two groups, set comparison and spatially-aware clusterings.

Set comparison distance measures look at the actual cluster *labeling* for each data point. Metrics under this category were conceived to evaluate any

clustering against the True solution. Within this category, two subgroups can be distinguished. The first one belongs to the *membership or counting-pairs group* that is directly calculated based on the confusion matrix. Well-established metrics include Rand Index (RI) [159] and Jaccard's Index [167]. Adjusted Rand Index [161] proposes a modification of RI to delimit value range to (0,1). The second subgroup within the set comparison category falls under the information theoretic metrics. They use the concept of entropy to compare clusterings, and include two main metrics called Variation of Information [158] and Mutual Information [156, 162]. These metrics also lack some properties that would enhance their interpretability. Based on original metrics, several variations have been proposed, such as Normalized Variation of Information [158], which normalizes through dividing by the upper bound $logn$ (where n is the total number of data points), or Normalized Mutual Information [162]. Apart from main subcategories (membership and information-theoretic), there are attemps to generalize over the existing corpus of metrics. Xiang et al. [168] proposed modeling the relationship between two clusterings as a bipartite graph, and derive distance metrics from this model. For a comprehensive review of set comparison metrics, we refer the interested reader to [157], which includes a complete list of metrics up to 2003, and to [163] that provide a thorough comparison of 32 membership measures from the literature.

While set comparison measures are quite popular, they ignore information about the spatial distribution of points within clusters, and so they are unable to differentiate between partitions that might be significantly dissimilar [169]. That is, instead of accounting only for the data point-to-cluster assigments, a spatially-aware metric might also appeal to the cluster shapes, the locations of the points in each cluster and the spacial relations among the clusters *CDistance*. The basic approach consists of modeling each cluster as a distribution and then measuring the distance between the distributions, e.g. by applying Mallows distance [170] or transportation distance (*CDistance* [171], LiftEMD [172]). We believe in the potential of spatially-aware metrics, in contrast to set comparison category, as they are applicable to a more extensive set of clustering problems (e.g. they usually do not require the same set of points in both clusterings) and they account for not only labeling of the final clusterings. However, most existing distance metrics under this category also suffer from high computational complexity: for example, *CDistance* has a worst case computational complexity of $O(n^3 log(n))$ [171]; or *ADCO* [160, 173] that considers all permutations among cluster matchings, has a computational complexity of $O(n!)$, being n the total number of clusters.

The goal of this manuscript is to define a set of distance measures with three main requirements: (1) they should not impose any restrictions over the clusterings [157], (2) the computational cost should be low, (3) but yet provide a useful score of clustering distance by taking into account the spatial layout of the partitions.

Next section describes the details of four alternative distance measures that fall under the spatially-aware category.

8.3 Formal Definition of Distance Measures

This section contains the definition of four different distance measures[1] that evaluate the dissimilarity between *any* two clusterings. We will first define the notation used throughout the rest of the manuscript.

Let $\mathcal{D}_1$ and $\mathcal{D}_2$ be two data sets composed of data points such that $\mathcal{D}_1 = \{x_1, x_2, \ldots, x_{|\mathcal{D}_1|}\}$ and $\mathcal{D}_2 = \{y_1, y_2, \ldots, y_{|\mathcal{D}_2|}\}$. Partition $\mathcal{P}_1$ groups data points from data set $\mathcal{D}_1$ into a set of clusters $\mathcal{P}_1 = \{A_1, A_2, \ldots, A_r\}$. Similarly, a second partition $\mathcal{P}_2$ is defined for data set $\mathcal{D}_2$ as $\mathcal{P}_2 = \{B_1, B_2, \ldots, B_t\}$.

Some of the distance measures use the centroid and/or standard deviation that can be computed for each cluster. A centroid is calculated as the mean for all data points contained in that cluster, such that $a_i = \bar{x}_k \in A_i$, and $b_j = \bar{y}_l \in B_j$. Associated to each cluster, we can also calculate its standard deviation σ_{a_i} and σ_{b_j}. The centroid and standard deviation for a cluster A_i at the v-nth dimension are denoted as $a_{i,v}$ and $\sigma_{a_{i,v}}$, respectively.

Our proposed measures deal with the following restrictions: (1) the intersection between $\mathcal{D}_1$ and $\mathcal{D}_2$ might be total, partial or none; (2) the number of data points per data set $|\mathcal{D}_1|$ and $|\mathcal{D}_2|$ might be different; and (3) the number of clusters per partition r and t might be different. We only make the following assumption: Both data sets must contain the same set of attributes[2]. For simplicity, attributes are continuous variables such that $\mathcal{D}_1, \mathcal{D}_2 \subseteq \mathbb{R}^n$.

We will define four alternative distance measures $D(\mathcal{P}_1, \mathcal{P}_2)$ that quantify the dis-similarity between any two clustering solutions or partitions $\mathcal{P}_1$ and $\mathcal{P}_2$. All of them fit in the spatially-aware category previously defined in Section 8.2.

Centroid-to-centroid distance (cc) The general idea behind this distance is to map each cluster from a partition to the *closest* cluster from the other partition and finally sum all those distances. For this particular case, the closeness between clusters is determined by the euclidean distance between their centroids.

Given $\{a_1, a_2, \ldots, a_r\}$ as the centroids for clusters $\{A_1, A_2, \ldots, A_r\}$, respectively, and $\{b_1, b_2, \ldots, b_t\}$ as the centroids for $\{B_1, B_2, \ldots, B_t\}$, the distance

[1] The *Distance measure* term already implies that large values (close to 1) indicate highly dissimilar partitions, while measures close to 0 refer to similar clusterings.

[2] The goal of comparing two clusterings relies on the implicit assumption that both sets of instances come from the same population, that is, all instances are objects characterized by a certain number of characteristics. Comparing two instances of different nature, or just described by another set of attributes would not make sense by nature. Therefore, our restriction is simply an implicit assumption of any correct comparison.

between a cluster A_i from $\mathcal{P}_1$ and a cluster B_j from $\mathcal{P}_2$ is defined as the distance between their centroids:

$$d'(A_i, B_j) = d'(B_j, A_i) := d(a_i, b_j) \quad \forall i = 1, \ldots, r \text{ and } \forall j = 1, \ldots, t$$

Each cluster in $\mathcal{P}_1$ is associated to its closest cluster in $\mathcal{P}_2$, that is, $\arg\min_{B_j \in \mathcal{P}_2} d'(A_i, B_j)$; and similarly for each cluster in $\mathcal{P}_2$, $\arg\min_{A_i \in \mathcal{P}_1} d'(A_i, B_j)$. The sum D' of all these distances is expressed as follows:

$$D'(\mathcal{P}_1, \mathcal{P}_2) := \sum_{i=1}^{r} \min_{j=1,\ldots,t} d'(A_i, B_j) + \sum_{j=1}^{t} \min_{i=1,\ldots,r} d'(A_i, B_j)$$

Finally, the distance is normalized, dividing by the maximum of all distances in $d'(A_i, B_j)$. The centroid-to-centroid distance between partitions $\mathcal{P}_1$ and $\mathcal{P}_2$ is defined as:

$$D_{cc}(\mathcal{P}_1, \mathcal{P}_2) := \frac{1}{(r+t) \max\limits_{\substack{i=1,\ldots,r \\ j=1,\ldots,t}} d'(A_i, B_j)} D'(\mathcal{P}_1, \mathcal{P}_2) \tag{8.1}$$

Data-to-centroid distance (dc) In contrast to previous distance, each data point in the data set is mapped separately to its closest centroid from the other partitions. Data-to-centroid distance computes the minimum euclidean distance from the items in one data set to the centroids in the other partition, and vice versa.

Given $\mathcal{D}_1 = \{x_1, x_2, \ldots, x_u\}$ and $\{b_1, b_2, \ldots, b_t\}$ as the centroids for $\mathcal{P}_2$, the distance $\tilde{d}$ between each data item in $\mathcal{D}_1$ to clustering $\mathcal{P}_2$ is defined as the minimum euclidean distance to its centroids; and similarly for each data item in $\mathcal{D}_2$ and clustering $\mathcal{P}_1$:

$$\tilde{d}(x_k, \mathcal{P}_2) := \min_{j=1,\ldots,t} d(x_k, b_j) \; \forall k = 1, \ldots, |\mathcal{D}_1|$$
$$\tilde{d}(y_l, \mathcal{P}_1) := \min_{j=1,\ldots,r} d(y_l, a_j) \; \forall l = 1, \ldots, |\mathcal{D}_2|$$

Note that the distance $\tilde{d}$ between data set $\mathcal{D}_1$ and clustering $\mathcal{P}_2$ is only looking at the actual data points, and completely ignoring the original clusters of clustering $\mathcal{P}_2$.

Indeed, Data-to-centroid distance does not perform a cluster-to-cluster mapping. Instead, this distance looks at the individual data points of a data set, and selects the closest centroid from the other clustering to each data point. More formally, for each data point x_k in $\mathcal{D}_1$, the distance uses the closest centroid in clustering $\mathcal{P}_2$, (and the same for dataset $\mathcal{D}_2$ and clustering $\mathcal{P}_1$). Two data points

x_i and x_j in in $\mathcal{D}_1$ might belong to the same cluster in $\mathcal{P}_1$, but each data point might be associated to a different cluster in $\mathcal{P}_2$. The distance is defined as follows:

$$\tilde{D}(\mathcal{P}_1, \mathcal{P}_2) := \sum_{k=1}^{|\mathcal{D}_1|} \tilde{d}(x_k, \mathcal{P}_2) + \sum_{l=1}^{|\mathcal{D}_2|} \tilde{d}(y_l, \mathcal{P}_1)$$

As in the case of centroid-to-centroid measure, we normalize by dividing by the maximum of all distances in $d(x_i, b_j)$ and $d(y_k, a_l)$:

$$D_{dc}(\mathcal{P}_1, \mathcal{P}_2) := \frac{\tilde{D}(\mathcal{P}_1, \mathcal{P}_2)}{(|\mathcal{D}_1| + |\mathcal{D}_2|)\ \max\left\{\max_{\substack{i=1,\dots,r \\ k=1,\dots,|\mathcal{D}_1|}} d(x_k, b_j),\quad \max_{\substack{j=1,\dots,t \\ l=1,\dots,|\mathcal{D}_2|}} d(y_l, a_i)\right\}} \tag{8.2}$$

Fitted-items distance (fi) This distance counts how many data points from a data set *fit* in the cluster set from the other partition. We use the term *fit* to denote that a data point falls within the range of a cluster.

A cluster can be defined as a rectangle of n dimensions (known as n-orthotope or hyperrectangle), where the center point is given by the cluster centroid, and the radius of each dimension is given by its standard deviation σ. A data point from data set $\mathcal{D}_1$ belongs to a cluster in $\mathcal{P}_2$ if the distance to its centroid is within the radius for each dimension. In order to extend the *range* of each cluster to cover 95% of the data, we will use twice the standard deviation as the value for radius (according to two- or three-sigma rule [174]).

We define the distance between each cluster A_i in $\mathcal{P}_1$ and each cluster B_j in $\mathcal{P}_2$ as the number of data points belonging to A_i that fit in cluster B_j:

$$\dot{d}(A_i, B_j) := |\left\{x \in A_i :\ \forall_{v=1}^{n} |x_v - b_{j,v}| < 2\sigma_{b_{j,v}}\right\}|$$
$$\dot{d}(B_j, A_i) := |\left\{y \in B_j :\ \forall_{v=1}^{n} |y_v - a_{i,v}| < 2\sigma_{a_{i,v}}\right\}|$$

Each cluster in $\mathcal{P}_1$ is mapped to the cluster in $\mathcal{P}_2$ that covers the maximum number of data points. So, the distance between $\mathcal{P}_1$ and $\mathcal{P}_2$ is proportional to the number of data points from $\mathcal{D}_1$ that fit in partition $\mathcal{P}_2$. Similarly, we can calculate the opposite distance from $\mathcal{D}_2$ and partition $\mathcal{P}_1$.

$$\dot{D}(\mathcal{P}_1, \mathcal{P}_2) := 1 - \frac{\sum_{i=1}^{r} \max_{j=1,\dots,t} \dot{d}(A_i, B_j)}{|\mathcal{D}_1|}$$

$$\dot{D}(\mathcal{P}_2, \mathcal{P}_1) := 1 - \frac{\sum_{j=1}^{t} \max_{i=1,\dots,r} \dot{d}(B_j, A_i)}{|\mathcal{D}_2|}$$

The fitted-items distance is expressed as the mean distance from $\mathcal{P}_1$ and $\mathcal{P}_2$ and vice versa:

$$D_{fi}(\mathcal{P}_1, \mathcal{P}_2) := \frac{\dot{D}(\mathcal{P}_1, \mathcal{P}_2) + \dot{D}(\mathcal{P}_2, \mathcal{P}_1)}{2} \tag{8.3}$$

Overlapping-volume distance (ov) This distance checks for the overlapping space (or intersection) between the n-volume covered by both partitions. Each cluster is modeled as a hyperrectangle. Then, n-volume of a cluster can be calculated as the product for all its n dimensions $\prod_{v=0}^{n} 2\sigma_v$, where σ_v is the standard deviation in v-th dimension.

The intersection of two clusters A_i and B_j is a hyperrectangle than can be calculated based on cluster centroids and standard deviations a_i, σ_{a_i} and b_j, σ_{b_j}. We first define $s_{i,j,v}$ as the *length* of the overlapping hyperrectangle between clusters A_i and B_j in v-th dimension:

$$s_{i,j,v} = \max\left\{0, (\min\left\{(a_i + \sigma_{a_i,v}), (b_j + \sigma_{b_j,v})\right\} - \max\left\{(a_i - \sigma_{a_i,v}), (b_j - \sigma_{b_j,v})\right\}\right\}$$

The distance between a pair of clusters A_i and B_j is equivalent to the n-volume of the overlapping hyperrectangle:

$$\check{d}(A_i, B_j) = \check{d}(B_j, A_i) := \prod_{v=1}^{n} s_{i,j,v}$$

The distance between two clusterings $\mathcal{P}_1$ and $\mathcal{P}_2$ is computed as the total overlapping n-volume of each pair of clusters A_i and B_j:

$$\check{D}(\mathcal{P}_1, \mathcal{P}_2) := \sum_{i=1}^{r} \sum_{j=1}^{t} \check{d}(A_i, B_j)$$

Finally, we normalize by the total volume covered by partitions $\mathcal{P}_1$ and $\mathcal{P}_2$:

$$D_{ov}(\mathcal{P}_1, \mathcal{P}_2) := 1 - \frac{2 * \check{D}(\mathcal{P}_1, \mathcal{P}_2)}{\sum_{i=0}^{r} \prod_{v=0}^{n} 2\sigma_{a_i,v} + \sum_{j=0}^{t} \prod_{v=0}^{n} 2\sigma_{b_j,v}} \qquad (8.4)$$

8.4 Distance Measure Validation

Before we perform any measure validation, one should answer the following question: What makes two clusterings *more* different? There is no single answer for that question. Therefore, evaluating the *quality* of any similarity measure will be very subjective, as there is no standard function which accurately measures how different two clusterings are [163]. Furthermore, the interpretation of similarity highly depends on the final goal of the comparison.

We cannot address the subjectivity of clustering comparison, but it is possible to alleviate the issue by reinforcing certain properties in any distance measure that quantifies the difference between clusters. First of all, a distance measure needs to be *strong* and comply with certain minimal properties (attached to metric

definition). Current section identifies which properties are satisfied mathematically. Second, Horta and Campello [163] proposes some desirable properties that any measure distance for clustering comparison might have. A series of empirical experiments have been conducted to evaluate our proposed distance metrics, and also other metrics existing in the literature.

8.4.1 Minimal Properties for Metric Definition

Any *metric* should comply with the following set of minimal properties. In a less restrictive manner, a *distance measure* only needs to fulfill the first two [157]:

- Positivity. $D(\mathcal{P}_1, \mathcal{P}_2) \geq 0$
- Symmetry. $D(\mathcal{P}_1, \mathcal{P}_2) = D(\mathcal{P}_2, \mathcal{P}_1)$
- Identity of indiscernibles. $D(\mathcal{P}_1, \mathcal{P}_2) = 0 \iff \mathcal{P}_1 = \mathcal{P}_2$
- Triangle inequality. $D(\mathcal{P}_1, \mathcal{P}_2) + D(\mathcal{P}_2, \mathcal{P}_3) \geq D(\mathcal{P}_1, \mathcal{P}_3)$

By definition, all four measures comply with the positivity and symmetry properties, so we can denote them as *distance measures*. Additionally, it is possible to prove the identify of indiscernibles property for every distance under some restrictions that are defined below.

8.5 Formal Validation of Metric Properties

A metric is a function that defines a distance between each pair of elements of a set (e.g. two partitions) and it must comply with several minimal properties that are defined as follows:

(a) Positivity. $D(\mathcal{P}_1, \mathcal{P}_2) \geq 0$

(b) Symmetry. $D(\mathcal{P}_1, \mathcal{P}_2) = D(\mathcal{P}_2, \mathcal{P}_1)$

(c) Identity of indiscernibles. $D(\mathcal{P}_1, \mathcal{P}_2) = 0 \iff \mathcal{P}_1 = \mathcal{P}_2$

(d) Triangle inequality. $D(\mathcal{P}_1, \mathcal{P}_2) + D(\mathcal{P}_2, \mathcal{P}_3) \geq D(\mathcal{P}_1, \mathcal{P}_3)$

In a less restrictive manner, a *distance measure* only needs to fulfill the first two.

8.5.1 Positivity and Symmetry

By definition, all proposed measures comply with the positivity and symmetry properties. This is enough to conclude that we have defined four valid *distance measures* for clustering comparison.

8.5.2 Identity of indiscernibles

It is possible to prove the identity of indiscernibles property under some restrictions. We will first define the following set of reasonable assumptions: each partition must have at least two clusters (thus, two data points) and there cannot be repeated centroids in each clustering:

$$\begin{array}{l} |\mathcal{P}_1| = r \geq 2, \quad |\mathcal{P}_2| = t \geq 2. \\ \forall a_i, a_j \in \mathcal{P}_1, j \neq i \implies a_i \neq a_j \\ \forall b_i, b_j \in \mathcal{P}_2, j \neq i \implies b_i \neq b_j \end{array}$$

The formal proof for the property is described separately for each distance:

Centroid-to-centroid distance We cannot prove that the clusterings are equal $\mathcal{P}_1 = \mathcal{P}_2$. Instead, we will focus on a less restrictive, but still valid supposition that the centroids are equal $\{a_1, a_2, \ldots, a_r\} = \{b_1, b_2, \ldots, b_t\}$.

Given $D_{cc}(\mathcal{P}_1, \mathcal{P}_2)$ definition:

$$D_{cc}(\mathcal{P}_1, \mathcal{P}_2) = 0 \implies D'(\mathcal{P}_1, \mathcal{P}_2) = 0$$

Now, based on $D'(\mathcal{P}_1, \mathcal{P}_2)$ definition and given that $\min_{j=1,\ldots,t} d'(A_i, B_j) \geq 0$ and $\min_{i=1,\ldots,r} d'(A_i, B_j) \geq 0$, we conclude that:

$$\sum_{i=1}^{r} \min_{j=1,\ldots,t} d'(A_i, B_j) + \sum_{j=1}^{t} \min_{i=1,\ldots,r} d'(A_i, B_j) = 0 \implies \begin{array}{l} \forall i \;\; \exists j : d'(A_i, B_j) = 0 \\ \forall j \;\; \exists i : d'(A_i, B_j) = 0 \end{array}$$

We can affirm that for each pair of clusters A_i, there is at least a cluster B_j for which the euclidean distance is equal to zero (and vice versa). If $d'(A_i, B_j) = 0$, then $d(a_i, b_j) = 0$, which means that centroids are equal $a_i = b_j$.

Given our previous assumptions that there are no repeated centroids, we can conclude that:

$$\{a_1, a_2, \ldots, a_r\} = \{b_1, b_2, \ldots, b_t\}$$

Data-to-centroid distance Based on $D_{dc}(\mathcal{P}_1, \mathcal{P}_2)$ definition:

$$D_{dc}(\mathcal{P}_1, \mathcal{P}_2) = 0 \implies \tilde{D}(\mathcal{P}_1, \mathcal{P}_2) = 0$$

Given that $\tilde{d}(x_k, \mathcal{P}_2) \geq 0$ and $\tilde{d}(y_l, \mathcal{P}_1) \geq 0$, we can conclude that every distance $\tilde{d}$ must be equal to zero:

$$\sum_{k=1}^{|\mathcal{D}_1|} \tilde{d}(x_k, \mathcal{P}_2) + \sum_{l=1}^{|\mathcal{D}_2|} \tilde{d}(y_l, \mathcal{P}_1) = 0 \implies \begin{array}{l} \tilde{d}(x_k, \mathcal{P}_2) = 0 \;\; \forall k = 1, \ldots, |\mathcal{D}_1| \\ \tilde{d}(y_l, \mathcal{P}_1) = 0 \;\; \forall l = 1, \ldots, |\mathcal{D}_2| \end{array}$$

If the euclidean distance d is zero, it means that the data point is equal to the centroid:

$$\begin{array}{l} \tilde{d}(x_k, \mathcal{P}_2) = 0 \implies \forall k, \exists j : d(x_k, b_j) = 0 \implies x_k = b_j \\ \tilde{d}(y_l, \mathcal{P}_1) = 0 \implies \forall l, \exists i : d(y_l, a_i) = 0 \implies y_l = a_i \end{array}$$

In conclusion, each data point x_k from data set $\mathcal{D}_1$ must be equal to a centroid b_j from partition $\mathcal{P}_2$; and each y_l must be equal to a centroid a_i.

$$\begin{array}{l} \{x_1, x_2, \ldots, x_{|\mathcal{D}_1|}\} = \{a_1, a_2, \ldots, a_r\} \\ \{y_1, y_2, \ldots, y_{|\mathcal{D}_2|}\} = \{b_1, b_2, \ldots, b_t\} \end{array}$$

Fitted-Items distance Similarly as in the previous cases, we start from the $D_{fi}(\mathcal{P}_1, \mathcal{P}_2)$ definition:

$$D_{fi}(\mathcal{P}_1, \mathcal{P}_2) = 0 \implies \begin{array}{l} \dot{D}(\mathcal{P}_1, \mathcal{P}_2) = 0 \\ \dot{D}(\mathcal{P}_2, \mathcal{P}_1) = 0 \end{array}$$

$\dot{D}(\mathcal{P}_1, \mathcal{P}_2)$ is equal to zero when all the data points in $\mathcal{D}_1$ are covered by partition $\mathcal{P}_2$. In other words, that all data points for each cluster A_i must fit in at least one cluster B_j. The reasoning is similar for the opposite case $\dot{D}(\mathcal{P}_2, \mathcal{P}_1)$:

$$\sum_{i=1}^{r} \max_{j=1,\ldots,t} \dot{d}(A_i, B_j) = |\mathcal{D}_1| \implies \forall i \quad 1, \ldots, t \quad \exists j : \dot{d}(A_i, B_j) = |A_i|$$

Regarding the identity of indiscernibles, we can conclude that for every A_i, all the data points fall within $2\sigma_{b_j}$, and for every B_j, all the data points fall within $2\sigma_{a_i}$:

$$\begin{array}{l} |\left\{x \in A_i : \quad \forall_{v=1}^{n} |x_v - b_{j,v}| < 2\sigma_{b_{j,v}}\right\}| = |A_i| \\ |\left\{y \in B_j : \quad \forall_{v=1}^{n} |y_v - a_{i,v}| < 2\sigma_{a_{i,v}}\right\}| = |B_j| \end{array}$$

Overlapping-volume distance $D_{ov}(\mathcal{P}_1, \mathcal{P}_2) = 0$ means that there is a complete overlap between partitions $\mathcal{P}_1$ and $\mathcal{P}_2$. Intuitively, we can conclude that the intersection between partitions $\check{D}(\mathcal{P}_1, \mathcal{P}_2)$ must be equal to the area covered by each partition:

$$\check{D}(\mathcal{P}_1, \mathcal{P}_2) = \sum_{i=0}^{r} \prod_{v=0}^{n} 2\sigma_{a_i,v} = \sum_{j=0}^{t} \prod_{v=0}^{n} 2\sigma_{b_j,v}$$

In summary, this is what we can conclude for each of the distances:

For centroid-to-centroid distance, we can conclude that if $D_{cc}(\mathcal{P}_1, \mathcal{P}_2) = 0$, then the centroids from both clusterings are equal:

$$\{a_1, a_2, \ldots, a_r\} = \{b_1, b_2, \ldots, b_t\}$$

In the case of data-to-centroid distance, $D_{dc}(\mathcal{P}_1, \mathcal{P}_2) = 0$ implies that all data points from data set $\mathcal{D}_1$ are the centroids for partition $\mathcal{P}_1$ and vice versa.

$$\begin{array}{l} \{x_1, x_2, \ldots, x_{|\mathcal{D}_1|}\} = \{a_1, a_2, \ldots, a_r\} \\ \{y_1, y_2, \ldots, y_{|\mathcal{D}_2|}\} = \{b_1, b_2, \ldots, b_t\} \end{array}$$

For fitted-items distance, we can only affirm that for every cluster A_i, there is at least one cluster B_j that *fits* all data points contained in A_i:

$$\begin{array}{l} |\left\{x \in A_i : \quad \forall_{v=1}^{n} |x_v - b_{j,v}| < 2\sigma_{b_{j,v}}\right\}| = |A_i| \\ |\left\{y \in B_j : \quad \forall_{v=1}^{n} |y_v - a_{i,v}| < 2\sigma_{a_{i,v}}\right\}| = |B_j| \end{array}$$

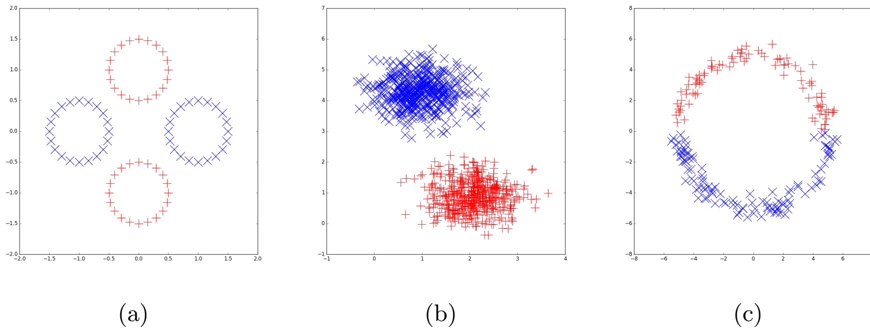
Finally, when overlapping-volume distance is equal to zero $D_{ov}(\mathcal{P}_1, \mathcal{P}_2) = 0$, it means that both partitions cover the same n-volume, i.e. the intersection between partitions must be equal to the area covered by each partition:

$$\check{D}(\mathcal{P}_1, \mathcal{P}_2) = \sum_{i=0}^{r} \prod_{v=0}^{n} 2\sigma_{a_i,v} = \sum_{j=0}^{t} \prod_{v=0}^{n} 2\sigma_{b_j,v}$$

8.5.3 Desirable Properties for Distance Measures

The applicability of any distance measure for clustering comparison can be extended if it complies with the following set of desirable properties: baseline, maximum, contrast, discriminant and low computational cost. All first four were proposed by Horta and Campello [163]. Intuitively, a distance measure should be able to detect identical partitions (baseline property) and completely different clusterings (maximum). Ideally, distance values should be *worse* (higher) for partitions that are progressively further from a reference one (contrast property). In some scenarios, we would like to find the *optimal* partition (discriminant property).

We have defined a set of three experiments that illustrate the behavior of measures under the previously defined properties. The first experiment (named Orthogonal clusters) aims to test the contrast, baseline and maximum properties. The Discriminant property can be evaluated by comparing the optimal solution against randomized clusters. Finally, the execution time of the distance measures can be checked by varying the data set size and number of clusters.



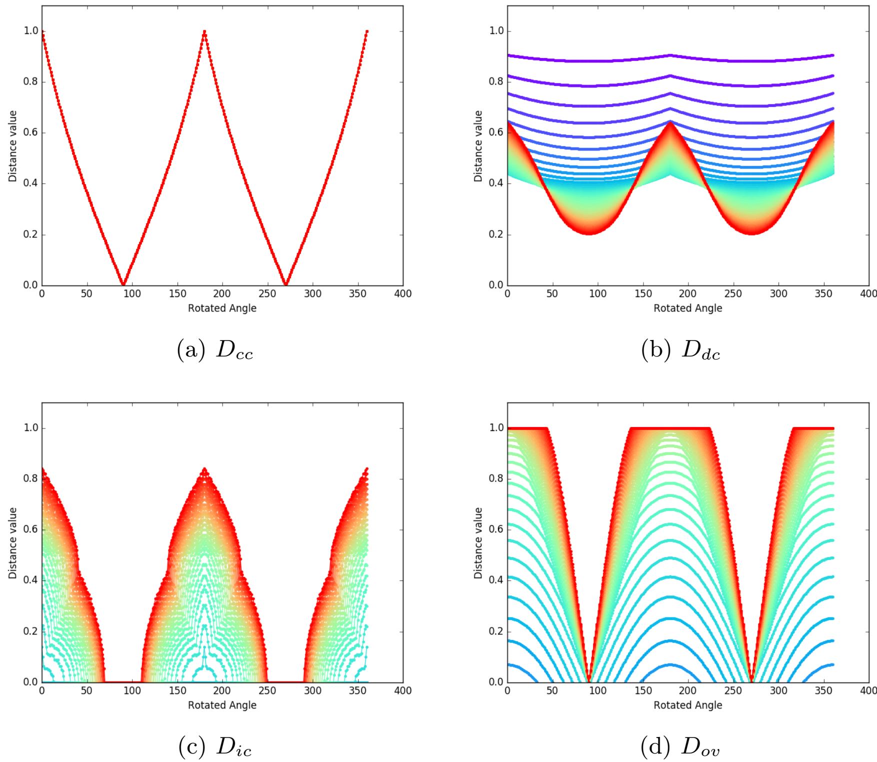
(a) (b) (c)

Figure 8.1: Sample data sets used in experiments: (a) Orthogonal clusters, (b) Random clusters and Compputational cost, (c) Comparison against other measures

Orthogonal clusters This experiment compares clusterings that are progressively *further* away. The first data set is composed of two clusters whose centroids are located on the horizontal axes $(0, -p)$ and $(p, 0)$. The centroids of second data set are on vertical axes, so centroids are $(0, -p)$ and $(0, p)$. The first data set is static, but the data points in the second data set will be rotated at increasing angles with respect to the center point (see Figure 8.1a). For angle values 0 and 180° degrees, the centroids are completely orthogonal, so the expected distance between clusterings should show the maximum value (1 or close to 1). At 90 and 270° angles, the centroids of both data sets will match, which means reaching the baseline value (0 in our case). We will also vary the value p, that indicates the distance between centroids.

Randomized clusters The reference partition is compared against progressively more *randomized* clusterings. We will introduce randomness in the reference partition (see Figure 8.1b) by shuffling part of the labels in the dataset (from 0 to 99%). In the resulting randomized partition, some data points will be reassign to a random cluster that is *not* the closest one. Results are the average distance between the reference clustering, and 100 re-shuffled data sets.

Computational cost The computational cost of proposed metrics depends on the number of clusters and/or the data set sizes. We will plot the execution time for varying the number of data points (the number of clusters is constant), and varying the number of clusters. Synthetic data sets are generated based on Gaussian distributions (see Figure 8.1b). Each measured time is the mean of 10 runs. Note that the execution time includes the computation of both centroids



(a) D_{cc} (b) D_{dc}

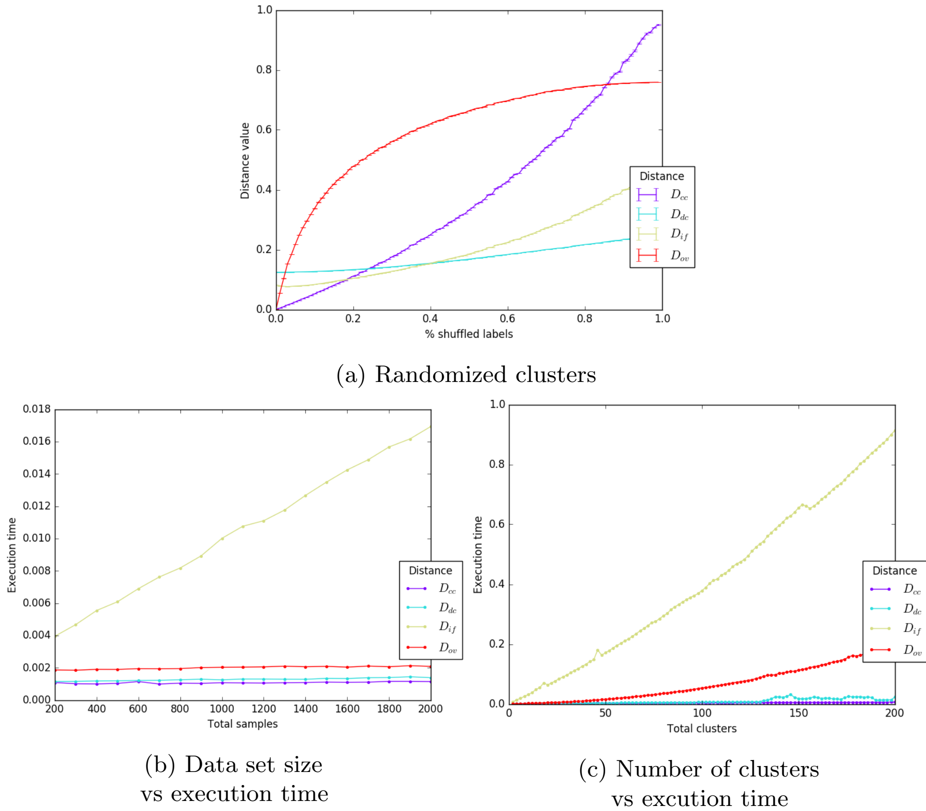
(c) D_{ic} (d) D_{ov}

Figure 8.2: Results for Orthogonal Cluster experiment

and standard deviation for each cluster. All distances have been implemented in Python v2.7 and experiments have been executed on a commodity desktop computer.

Discussion of results

Results for the Orthogonal Clusters experiment are shown in Figure 8.2. All proposed distances comply with the contrast property: they show progressively higher values for more distant clusterings. Baseline and maximum properties can also be seen in the Orthogonal clusters experiment. All distances (except data-to-centroid) reach value 0 for data sets with matching centroids and value 1 for data sets with orthogonal centroids. We could easily fix the baseline value for data-to-centroid distance, e.g. normalizing by the sum of minimum distances to



(a) Randomized clusters

(b) Data set size vs execution time

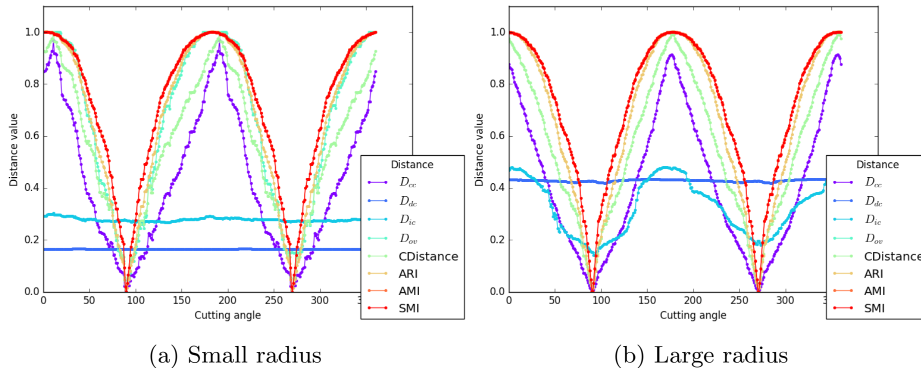
(c) Number of clusters vs excution time

Figure 8.3: Results for Randomized clusters and Computational cost experiments

each cluster.

Another important conclusion can be extracted by looking at *when* measure distances reach value 1. Fitted-items and overlapping-volume distances focus on comparing the data sets, and value 1 means that there is no overlap among data sets. In contrast, centroid-to-centroid and data-to-centroid distances are designed to measure how far apart are the clusterings from each other. Thus, we could identify two subtypes of measures, a first set that focuses on the *shape* of the clusterings, i.e. that measures the distance between data sets (no overlap means that data sets have nothing in common); and a second set of measures that look at the *position* of the clusters.

Figure 8.3a shows that all distances are monotonic when comparing the reference clustering against a new solution with increased percentage of randomized



(a) Small radius (b) Large radius

Figure 8.4: Results for comparison against other measures

items. We can conclude that all distances comply with the discriminant property.

Finally, we evaluate the execution time of proposed distances. The theoretical computational cost for our proposed metrics is $O(nk)$ for data-to-centroid and fitted-items distances, and $O(max(k^2, n/k))$ for centroid-to-centroid and overlapping-volume distances (given k clusters, and n data points). Figures 8.3b and 8.3c show the execution time against varying the number of data points and number of clusters. Empirical results match and even improve the theoretical results. The execution time of centroid-to-centroid and data-to-centroid distances is negligible, whereas fitted-items measure increases linearly with the number of data points, and with the number of clusters.

8.5.4 Comparison against other measures

We have chosen a representative sample from other metrics in the literature. Some of the most popular metrics in the clustering community are the Adjusted Rand Index (ARI) based on pair-counting, and the Adjusted Mutual Information (AMI) based on information-theoretical measures [162]. We have also included a more recent proposal by [175], called Standard Mutual Info (SMI). As a representative of spatially-aware metrics, we have selected CDistance [171] [3].

The experiment is designed to cover all range of values for every distance. Note that some of the selected distances impose the restriction of using the same data set. We have generated a synthetic set of points (see Figure 8.1c), that are located in a circumference. The reference clustering is fixed, and data points are

[3]We have used provided Matlab implementation for these metrics. The code for ARI, AMI and SMI can be found in `https://github.com/ialuronico/StandardizedMutualInformation`, and the code for CDistance in `https://biocomp.wisc.edu/data`.

Table 8.1: General description of datasets used in the experiments. Note that *Dataset* refers to the short name selected to identify the dataset while *full name* is the complete name that appears under UCI repository.

Dataset	Full name	#Instances	#Attributes	#Classes
Magic	MAGIC Gamma Telescope	19020	10	2
Breast cancer	Breast Cancer Wisconsin (Original)	683	10	2
Eye state	EEG Eye State	14980	14	2
HTRU	HTRU2	17898	8	2
Red wine	Wine Quality (red variant)	1599	11	6
White wine	Wine Quality (white variant)	4898	11	7

divided in two halves (clusters). At each step, the angle is modified, so a few data points are reassigned to the other cluster. This experiment resembles the idea behind Orthogonal clusters experiment.

Results are shown in Figure 8.4. All metrics from the literature (SMI, ARI, AMI and CDistance), together with our proposed centroid-to-centroid and overlapping-volume distances are able to capture the differences between two clusterings. The distance values are almost identical. However, SMI, ARI and AMI measures always assume the same data set. CDistance does not impose restrictions on the data sets or number of clusters, but it is based on the calculation of the optimal transportation distance with worst case of $O(n^3 log(n))$ [171].

As a general conclusion, centroid-to-centroid and overlapping-volume measures show competitive results compared to others existing in the literature, without imposing any restrictions in the clusterings and also at a very low computational cost (especially, centroid-to-centroid distance).

8.5.5 Experiments with real datasets

So far, experiments have been based on synthetic data. This section is devoted to validate our proposed distance measures on some real datasets, and contrast the results against some distances from the literature. For this purpose, we have selected a total of six datasets from UCI Repository[4], with varied number of instances, attributes and classes. Table 8.1 gathers details about each dataset.

Our proposed distance measures are, in many ways, less restrictive than others encountered in the literature. However, for the sake of comparison, we are forced to accommodate experiment design to the most restrictive distance measures. For that reason, experiments will be based on clustering trained on the *same* dataset. This section evaluates how our proposed distances are able to achieve similar

[4]UC Irvine Machine Learning Repository, `http://archive.ics.uci.edu/ml/`

results to other well-established distance measures, but yet, they are applicable to a larger number of problems (e.g. clusterings trained on different data sets) and with a lower computational cost.

Two types of experiments have been designed for that purpose: evaluating K-means clustering through training (iterations) and comparing clusterings obtained from different algorithms. We will use our four proposed distances, and for comparison sake, we will use AMI, ARI and SMI. (CDistance has not been included, as its processing time becomes untreatable.)

K-Means iterations K-means algorithm is applied on each of the data sets. This experiment stores the final clustering on each iteration (up to 50). Each of them is compared against the base clustering, that is, the clustering resulting of the first iteration that has been initialized with random cluster centers. Results are presented in Figure 8.5. The general conclusion is that our distances, equally to ARI and AMI distance measures, are able to capture the clustering variability. Note that the K-Means algorithm many times converges before the maximum number of iterations, so the distance values remain pretty constant. As an exception, SMI measure is not bounded within the range (0,1), which makes results harder to be evaluated.

Evaluating clusterings from different algorithms In this second experiment, we compare the dataset clustered using its original labels, vs the result of different clustering algorithms: Mini-batch KMeans (*minibatch*), K-Means (*kmeans*), Gaussian Mixture Model (*gmm*) and Birch model (*birch*). Table 8.2 gathers results from all algorithms, trained over all data sets. Let us start by looking at our proposed distances. We can quickly identify that D_{if} and D_{ov} distances show values close 1, meaning that selected dataset clusters have no overlapping among them. For this experiment, the most suitable distances are D_{cc} and D_{dc}, that show values ranging from 0 to 1. Minibatch, kmeans and also Birch algorithms show pretty homogeneous score values, while GMM algorithm usually shows a slightly higher or lower score (e.g. see White wine and HTRU datasets). The same behavior is present with AMI and ARI distance metrics (SMI once again leads to hardly interpretable scores). This result is really important, as it shows that our proposed distances can be applicable for the same task as state-of-art distance measures. However, ARI, AMI and SMI distances are calculated based on cluster labeling and thus are restricted to clusterings constructed with the same dataset. Our proposed distances do not impose this restriction. Although not included here, SMI and CDistance distances suffer from longer execution times. Additionally, SMI does not seem to comply with the discriminant property.

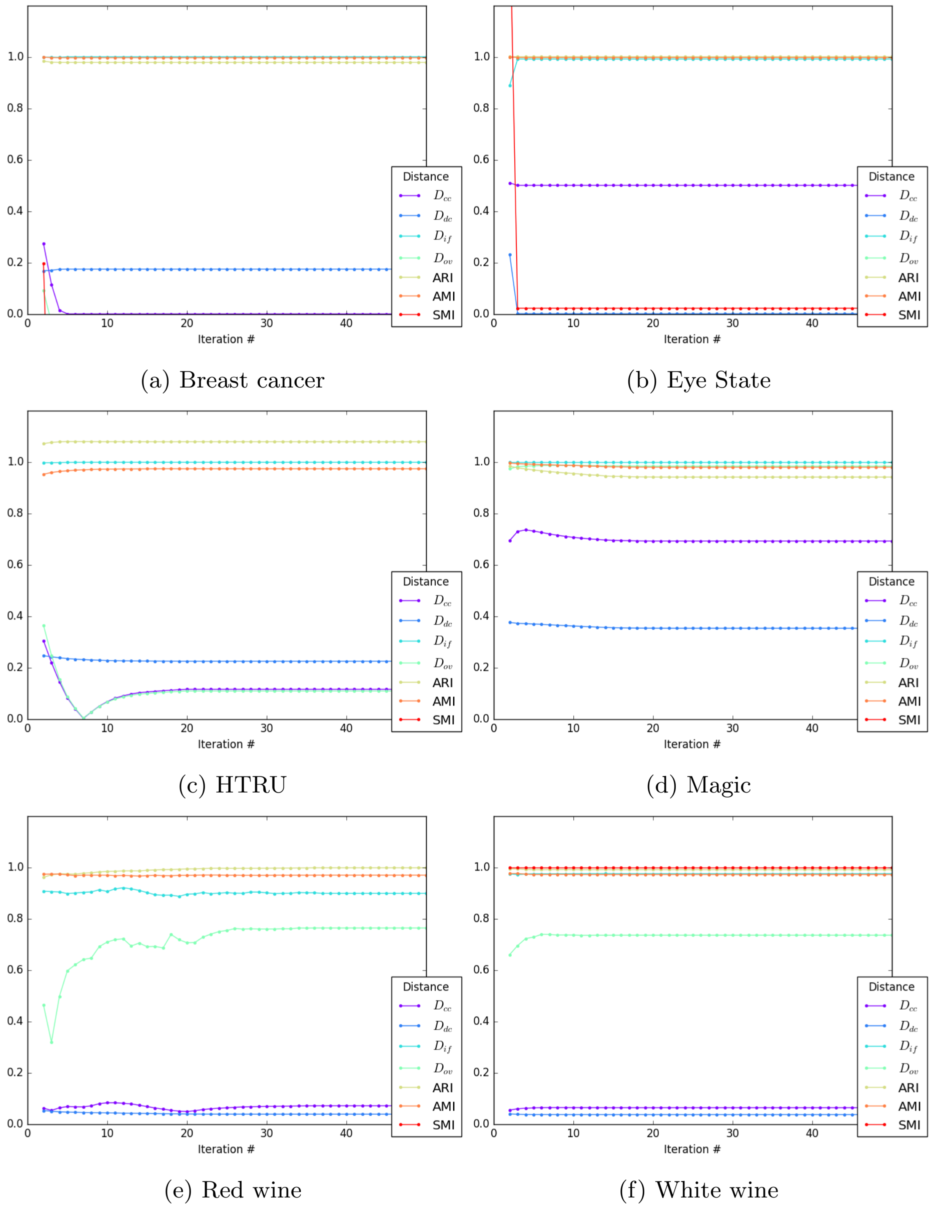
Table 8.2: Distance values from comparing original labeled datasets and clusterings resulting from different algorithms.

Dataset	Clustering Alg	D_{cc}	D_{dc}	D_{if}	D_{ov}	AMI	ARI	SMI
Magic	minibatch	0.6479	0.3765	1.0000	0.9743	0.9387	0.9777	-338.34
	kmeans	0.6570	0.3785	1.0000	0.9761	0.9406	0.9789	-322.95
	gmm	0.7640	0.4111	1.0000	0.9680	0.9379	0.9722	-483.16
	birch	0.6336	0.3740	1.0000	0.9714	0.9487	0.9828	-252.08
Red wine	minibatch	0.1322	0.0573	0.9162	0.9290	0.9966	0.9684	-18.754
	kmeans	0.1441	0.0396	0.8968	0.9250	1.0020	0.9623	-22.377
	gmm	0.1544	0.0456	0.9174	0.9207	0.9686	0.9507	-7.320
	birch	0.1482	0.0409	0.9087	0.9141	1.0098	0.9622	-22.083
Eye State	minibatch	0.5192	0.0993	0.9949	1.0000	1.0000	0.9998	1.5898
	kmeans	0.5001	0.0007	0.9946	1.0000	1.0001	0.9974	0.0222
	gmm	0.5001	0.0005	0.9939	1.0000	1.0000	0.9982	0.8647
	birch	0.5001	0.0007	0.9946	1.0000	1.0001	0.9974	0.0222
White wine	minibatch	0.1053	0.0508	0.9660	0.9772	0.9902	0.9705	1.0000
	kmeans	0.1064	0.0512	0.9679	0.9762	0.9905	0.9715	1.0000
	gmm	0.0876	0.0684	0.9717	0.9606	0.9590	0.9303	1.0000
	birch	0.0996	0.0523	0.9627	0.9736	0.9885	0.9755	1.0000
HTRU	minibatch	0.6300	0.2844	0.9967	0.9970	1.0771	0.9611	-390.54
	kmeans	0.5891	0.2637	0.9977	0.9959	1.0779	0.9733	-239.23
	gmm	0.3931	0.3193	0.9945	0.9742	0.6576	0.7386	-2689.1
	birch	0.5833	0.2598	0.9978	0.9957	1.0764	0.9755	-212.84
Breast Cancer	minibatch	0.6687	0.2640	0.9978	0.9979	0.9781	0.9964	-1.0860
	kmeans	0.7019	0.2652	0.9985	0.9987	0.9781	0.9964	-1.0860
	gmm	0.5022	0.0419	0.9978	0.9982	1.0027	0.9974	0.6934
	birch	0.5022	0.0419	0.9978	0.9982	1.0027	0.9974	0.6934

8.6 Conclusions and Summary

Clustering comparison is a useful task with many applications in the clustering domain. A diversity of distance measures have been proposed in the literature for different goals (e.g. cluster evaluation or data stream clustering). Existing measures can be classified into two main categories: membership and spatially-aware metrics. We have proposed four different distance measures that fall under the latter category, and that deal with typical limitations imposed by traditional measures: (1) different number of clusters, and (2) different (possibly disjoint) data sets. Centroid-to-centroid and data-to-centroid distances focus on the *position* of the clusterings, whereas fitted-items and overlapping-volume distances focus on the *shape* of the solutions. Measures have been validated both formally and empirically, under several properties. As a general conclusion, we have found that two distances (centroid-to-centroid and overlapping-volume distances) show competitive results in contrast to other measures in the literature, and very low computational requirements. Moreover, experiments on real datasets validate

that our proposed metrics can be applied to real-world tasks, equally to other state-of-art metrics (AMI, ARI, SMI). However, our metrics can solve a larger subset of problems as they do not impose so many restrictions.



(a) Breast cancer

(b) Eye State

(c) HTRU

(d) Magic

(e) Red wine

(f) White wine

Figure 8.5: Results for K-Means Iterations experiment. Each plot corresponds to a different real data set.

Chapter 9

Summary and Conclusions

This chapter summaries the main contributions and conclusions provided in this dissertation. Moreover, this chapter addresses a set of general directions for further research. More specific conclusions and future work lines have been exposed in each corresponding chapter.

9.1 Conclusions

Virtualization brings many benefits in the management of a data center, including a more fine-grained manageability of applications and tasks. However, the use of containers or VMs imposes several challenges in the general resource sharing.

In Chapter 3, we have introduced a set of useful models to characterize the whole data center, including the network infrastructure, the application layout and workload types.

Chapter 4 demonstrates that an IaaS provider can improve the VM placement policy in use by applying an optimization strategy, achieving benefits not only for the provider but also for the client and the end-user. Furthermore, this optimization can be done at a negligible cost: it is applied when allocating a new application, taking only a few seconds. Benefits for the provider are measured in terms of used servers and switches, and immediately translate into reduced power demands (resulting in a "greener" data center). Benefits for the applications are achieved by improving their ability to process user requests. As a direct effect of improving communication latencies, the average execution time per request is reduced (up to 11–19%). Thus, the cloud client (that is, the application owner) will be better able to comply with QoS expected by end users. Simultaneously,

using optimization we are able to improve the number of active servers in the data center and, therefore, the overall energy consumed: for highly loaded data centers, savings range between 7.26–13.81%. Globally, the best tested placement strategy is FF with SPEA-2 optimization, although in some instances (optimized) RR yields better results.

Chapter 5 focuses on the auto-scaling problem: The aim of the auto-scaler is to dynamically adapt the resources assigned to the elastic applications, depending on the input workload. This auto-scaler may be either an ad-hoc implementation for a particular application, or a generic service offered by the IaaS provider. In any case, the system should be able to find a trade-off between meeting the SLA of the application and minimizing the cost of renting cloud resources. We propose a taxonomy to classify state-of-art techniques into 5 categories: rules, control theory, time series analysis, queuing theory and reinforcement learning.

Rules with static thresholds are the most wide-spread auto-scaling technique among current cloud providers (e.g. Amazon AWS). In Chapter 6, we propose a variant named rules with dynamic thresholds that proves to be very effective when compared to some other representative auto-scalers from the literature.

In Chapter 7, we have presented an online algorithm for detecting the noisy-neighbor effect, that is, anomalies in resource usage caused by VMs or containers competing for the same shared resources. The strategy consists of comparing DPGMM models trained at different times. By scoring the similarity between models with metrics that compare probability distributions, we are able to correctly detect the presence of anomalies in the resource consumption. Even when focusing on a particular type of anomaly (noisy neighbor effect), the impact on resource usage greatly depends on the application. Our methodology is tested on four different applications: a static webpage, batch application, and Spark (in-memory), Memcached server, all deployed as Docker containers.

Clustering comparison is a useful task with many applications in the clustering domain. Our anomaly detection algorithm uses some novel distance metrics to compare GMMs. Derived by this work, Chapter 8 explores the comparison of hard clusterings and proposes a set of new distance metrics. Several experiments are carried out to proof their usability.

9.2 Future Research Directions

We identify several lines of research work:

9.2.1 Application placement

The utilization of this optimization-based placement requires an effort from the clients: they must specify, or at least provide some hints about, the application

architecture and its communication needs. This is an old and well-known problem: application dimensioning. Our proposal also requires including, in the decision processes for data center resource management, knowledge about the structure of its interconnection network.

The proposed solution could be easily extensible to other objective function: e.g. controlling the thermal effect in the data center. This work is focused on the initial VM placement of applications. However, the state of a data center is very dynamic: new applications arrive, other applications are removed, and deployed applications can increase/reduce the resources allocated to it. After all, one of the key characteristics of cloud computing is elasticity. Most probably, after some time, the initial VM placement may become sub-optimal (for example, using an excessive number of active servers), and reallocation of some VMs could be necessary (for example, to consolidate more VMs in fewer servers to obtain a smaller set of active servers). This process is called re-consolidation. We plan to address re-consolidation as a multi-objective optimization problem, trading off its benefits with its costs. Each VM migration should be planned carefully as it causes extra CPU costs and network overhead that may negatively affect the allocated applications.

Elasticity allows the applications to dynamically scale the acquired resources (the number of VMs in horizontal scaling) depending on the input workload. Thus, the number of VMs will vary with time and the infrastructure provider should be able to optimize not only the initial placement, but also the addition of new VMs. We plan to adapt our proposal to deal with scalable applications.

Providers usually over-subscribe resources: users rarely exploit 100% of the assigned resources (including cores, memory, network bandwidth, etc.) Therefore, it is common practice to assign to a server "extra" slots. However, providers have to monitor resource usage carefully, to ensure that the aggregated current demands do not exceed server capacity: this would be very negative in terms of the QoS perceived by clients. The QoS could be easily fitted as an extra objective in a multi-objective optimization formulation of the VM placement and/or re-consolidation.

9.2.2 Auto-scaling

Cloud computing environments allow users to dynamically scale their applications. The key problem is how to lease the right amount of Cloud computing environments allow users to dynamically scale their applications, on a pay-as-you-go basis. Application re-dimensioning can be implemented effortlessly, adapting the resources assigned to the application to the incoming user demand. However, the identification of the right amount or resources to lease in order to meet the required Service Level Objectives, while keeping the overall cost low, is not an

easy task. Many techniques have been proposed for automating this application scaling. We propose a classification of the techniques into five main categories: static threshold-based rules, control theory, reinforcement learning, queuing theory and time series analysis.

Using Q-learning algorithms to automate the scaling of each application and using a Neural network to deal with continuous state-spaces in actions and states.

9.2.3 Anomaly detection

For the current article, we have focused on the detection of performance anomalies in resource usage. More specifically, VM interference problem. This anomaly detection algorithm might be applicable to other problems and domains: e.g. detecting anomalies in a several tier application, by including the resource usage of each layer in the same instance.

The anomaly detection algorithm itself could be extended in several ways. Other approaches could be applied for pattern detection in the pre-alarm time series, to filter out false positives. We could leverage Temporally-Dependent Dirichlet Process Mixtures in order to capture time-related patterns. Time series prediction techniques could be applied to forecast anomalies in resource usage. In contrast to our current unsupervised approach, it would be possible to keep a dynamic store of known anomalies (as DPGMMs).

Finally, detecting the anomaly is not enough. We plan to develop a Q-learning based controller to re-distribute the resources among different VMs and resolve the anomaly.

9.2.4 Comparison metrics for clusterings

We plan to explore variants of the proposed measures: e.g. data-to-centroid distance is inspired by the objective function of K-Means algorithm, and similarly, many distances could be defined based on the objective function used by other clustering algorithms; fitted-items and overlapping-volume distances could be extended by varying the radius coefficient to increase the coverage of data items. We can also identify potentially useful properties in our distances: e.g. centroid-to-centroid distance is equal to 1 *only* when all clusters from a clustering are equidistant to the clusters in the other solution.

There are some partial attempts to provide basic formalization [157] and determining desirable properties for distance measures [163]. However, there is a lack of a standard generalized framework for metric definition and validation that could be a relevant line of research work.

Finally, we plan to explore the applicability of our proposed measures to different tasks, such as data stream clustering, ensemble clustering or re-clustering (i.e grouping sets of similar partitions, or model identification). This will help to

determine the suitability of each measure to a particular problem and to identify key characteristics to automate the measure selection.

Bibliography

[1] Tania Lorido-Botran, Sergio Huerta, Luis Tomas, Johan Tordsson, and Borja Sanz. An Unsupervised Approach to Online Noisy-Neighbor Detection in Cloud Data Centers. *Expert Systems With Applications*, 2017.

[2] Tania Lorido-Botrán, Sergio Huerta, and Borja Sanz. Towards Spatially-Aware Comparison of Hard Clusterings. 2017.

[3] Tania Lorido-Botran, Jose Miguel-Alonso, and Jose A. Lozano. A Review of Auto-scaling Techniques for Elastic Applications in Cloud Environments. *Journal of Grid Computing*, pages 1–34, 2014. ISSN 1570-7873. doi: 10.1007/s10723-014-9314-7.

[4] Jose A. Pascual, Tania Lorido-Botran, Jose Miguel-Alonso, and Jose A. Lozano. Towards a Greener Cloud Infrastructure Management using Optimized Placement Policies. *Journal of Grid Computing, Special Issue*, 2014. doi: 10.1007/s10723-014-9312-9.

[5] Tania Lorido-Botran, Jose A. Pascual, Jose Miguel-Alonso, and Jose A. Lozano. Optimization of Application Placement towards a Greener Cloud Infrastructure. In *EvoPar*, 2014.

[6] Tania Lorido-Botran, Jose Miguel-Alonso, and Jose Antonio Lozano. Comparison of Auto-scaling Techniques for Cloud Environments. In *Actas de las XXIV Jornadas de Paralelismo (SARTECO), CEDI*, 2013.

[7] Ian Foster, Yong Zhao, Ioan Raicu, and Shiyong Lu. Cloud computing and grid computing 360-degree compared. In *Grid Computing Environments Workshop, 2008. GCE'08*, pages 1–10. Ieee, 2008.

[8] Amazon Elastic Compute Cloud (Amazon EC2). `http://http://aws.amazon.com/ec2/`, 2012. [Online; accessed 15-May-2018].

[9] Rackspace. The open cloud company. `http://www.rackspace.com/`, 2012. [Online; accessed 15-May-2018].

[10] Google Compute Engine. `http://cloud.google.com/products/compute-engine.html/`, 2012. [Online; accessed 15-May-2018].

[11] AWS Elastic Beanstalk (beta). Easy to begin, Impossible to outgrow. `http://aws.amazon.com/elasticbeanstalk/`, 2012. [Online; accessed 15-May-2018].

[12] Google App Engine. `http://cloud.google.com/products/`, 2012. [Online; accessed 15-May-2018].

[13] Microsoft Windows Azure. https://www.windowsazure.com/en-us/, 2012. [Online; accessed 15-May-2018].

[14] Google Apps for Business. http://www.google.com/intl/es/enterprise/apps/business/products.html, 2012. [Online; accessed 15-May-2018].

[15] Microsoft Office 365. http://www.microsoft.com/en-us/office365/online-software.aspx, 2012. [Online; accessed 15-May-2018].

[16] Salesforce.com. http://www.salesforce.com/, 2012. [Online; accessed 15-May-2018].

[17] Google data center faq. http://www.datacenterknowledge.com/archives/2017/03/16/google-data-center-faq.

[18] Cisco global cloud index: Forecast and methodology, 2016–2021 white paper. https://www.cisco.com/c/en/us/solutions/collateral/service-provider/global-cloud-index-gci/white-paper-c11-738085.html.

[19] Inside amazon's cloud computing infrastructure. https://datacenterfrontier.com/inside-amazon-cloud-computing-infrastructure/.

[20] The facebook data center faq. http://www.datacenterknowledge.com/data-center-faqs/facebook-data-center-faq-page-2.

[21] Microsoft (msft) reports earnings beat, azure revenue soars 98 https://www.nasdaq.com/article/microsoft-msft-reports-earnings-beat-azure-revenue-soars-98-cm914225/.

[22] The social impact of cloud. https://www.forbes.com/sites/sap/2012/01/10/the-social-impact-of-cloud-2/{#}74d7a872575a, 2012.

[23] Cloud service models and most common cloud computing use cases. https://cloudacademy.com/blog/most-common-cloud-service-models/.

[24] How cloud computing is impacting everyday life. https://www.ibm.com/blogs/cloud-computing/2013/04/04/how-cloud-computing-is-impacting-everyday-life/.

[25] Rui Máximo Esteves and Chunming Rong. Social impact of privacy in cloud computing. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 593–596. IEEE, 2010.

[26] Peter Bodik, Moises Goldszmidt, Armando Fox, Dawn B Woodard, and Hans Andersen. Fingerprinting the datacenter: automated classification of performance crises. In *Proceedings of the 5th European conference on Computer systems*, pages 111–124. ACM, 2010.

[27] Charles Reiss, John Wilkes, and Joseph L Hellerstein. Google cluster-usage traces: format+ schema. *Google Inc., White Paper*, pages 1–14, 2011.

[28] Christina Delimitrou and Christos Kozyrakis. Quasar: resource-efficient and qos-aware cluster management. *ACM SIGPLAN Notices*, 49(4):127–144, 2014.

[29] James M Kaplan, William Forrest, and Noah Kindler. Revolutionizing data center energy efficiency. Technical report, Technical report, McKinsey & Company, 2008.

[30] Microsoft research project puts cloud in ocean for the first time. https://news.microsoft.com/features/microsoft-research-project-puts-cloud-in-ocean-for-the-first-time/.

[31] Peter Mell and Tim Grance. The NIST definition of cloud computing. 2011.

[32] Prateek Sharma, Lucas Chaufournier, Prashant Shenoy, and YC Tay. Containers and Virtual Machines at Scale: A Comparative Study. In *Proceedings of the 17th International Middleware Conference*, page 1. ACM, 2016.

[33] Eucalyptus, 2014. URL `http://www.eucalyptus.com/`. [Online; accessed 6-June-2014].

[34] OpenNebula, 2014. URL `http://opennebula.org/`. [Online; accessed 6-June-2014].

[35] NetIQ, 2014. URL `https://www.netiq.com/products/recon/`. [Online; accessed 6-June-2014].

[36] VMware, 2014. URL `http://www.vmware.com/products/capacity-planner/`. [Online; accessed 6-June-2014].

[37] IBM, 2014. URL `www.ibm.com/software/products/us/en/workload-deployer`. [Online; accessed 6-June-2014].

[38] P Fan, Z Chen, J Wang, and Z Zheng. Online Optimization of VM Deployment in IaaS Cloud. In *ICPADS*, pages 760–765, 2012.

[39] S Georgiou, K Tsakalozos, and A Delis. Exploiting Network-Topology Awareness for VM Placement in IaaS Clouds. In *CGC*, pages 151–158, 2013.

[40] V Mann, A Kumar, P Dutta, and S Kalyanaraman. VMFlow: leveraging VM mobility to reduce network power costs in data centers. In *NETWORKING - Volume I*, pages 198–211, 2011.

[41] Xiaoqiao Meng, Vasileios Pappas, and Li Zhang. Improving the Scalability of Data Center Networks with Traffic-aware Virtual Machine Placement. In *IEEE INFOCOM*, pages 1154–1162, March 2010. ISBN 978-1-4244-5836-3.

[42] T Wo, Q Sun, B Li, and C Hu. Overbooking-Based Resource Allocation in Virtualized Data Center. In *ISORCW*, pages 142–149, 2012.

[43] Tevfik Yapicioglu and Sema Oktug. A Traffic-Aware Virtual Machine Placement Method for Cloud Data Centers. In *Proceedings of the 2013 IEEE/ACM 6th International Conference on Utility and Cloud Computing*, UCC '13, pages 299–301, Washington, DC, USA, 2013. IEEE Computer Society. ISBN 978-0-7695-5152-4.

[44] Dzmitry Kliazovich, Pascal Bouvry, and SameeUllah Khan. DENS: data center energy-efficient network-aware scheduling. *Cluster Computing*, 16(1):65–75, 2013. ISSN 1386-7857.

[45] Weiwei Fang, Xiangmin Liang, Shengxin Li, Luca Chiaraviglio, and Naixue Xiong. VMPlanner: Optimizing virtual machine placement and traffic flow routing to reduce network power costs in cloud data centers. *Computer Networks*, 57(1): 179–196, 2013.

[46] N Tziritas, Cheng-Zhong Xu, T Loukopoulos, S U Khan, and Zhibin Yu. Application-Aware Workload Consolidation to Minimize Both Energy Consumption and Network Load in Cloud Environments. In *Parallel Processing (ICPP), 2013 42nd International Conference on*, pages 449–457, October 2013.

[47] Shao-Heng Wang, P.P.-W. Huang, C.H.-P. Wen, and Li-Chun Wang. EQVMP: Energy-efficient and QoS-aware virtual machine placement for software defined datacenter networks. In *Information Networking (ICOIN), 2014 International Conference on*, pages 220–225, February 2014.

[48] Jonathan Kupferman, Jeff Silverman, Patricio Jara, and Jeff Browne. Scaling into the cloud. Technical report, University of California, Santa Barbara; CS270 - Advanced Operating Systems, 2009.

[49] Bart Jacob, Richard Lanyon-Hogg, Devaprasad K Nadgir, and Amr F Yassin. *A practical guide to the IBM autonomic computing toolkit.* IBM, International Technical Support Organization, 2004.

[50] M Maurer, I Breskovic, V C Emeakaroha, and I Brandic. Revealing the MAPE loop for the autonomic management of Cloud infrastructures. In *Computers and Communications (ISCC), 2011 IEEE Symposium on*, pages 147–152, 2011.

[51] Khalid Alhamazani, Rajiv Ranjan, Karan Mitra, Fethi Rabhi, Samee Ullah Khan, Adnene Guabtni, and Vasudha Bhatnagar. An Overview of the Commercial Cloud Monitoring Tools: Research Dimensions, Design Issues, and State-of-the-Art. *arXiv preprint arXiv:1312.6170*, 2013.

[52] H Ghanbari, B Simmons, M Litoiu, and G Iszlai. Exploring Alternative Approaches to Implement an Elasticity Policy. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 716–723. IEEE, 2011.

[53] Ming Mao and Marty Humphrey. A Performance Study on the VM Startup Time in the Cloud. In *Proceedings of the 2012 IEEE Fifth International Conference on Cloud Computing*, CLOUD '12, pages 423–430, Washington, DC, USA, 2012. IEEE Computer Society.

[54] Cheng Huang, David A Maltz, Jin Li, and Albert Greenberg. Public DNS system and global traffic management. In *INFOCOM, 2011 Proceedings IEEE*, pages 2615–2623. IEEE, 2011.

[55] Xing Pu, Ling Liu, Yiduo Mei, Sankaran Sivathanu, Younggyun Koh, and Calton Pu. Understanding performance interference of i/o workload in virtualized cloud environments. In *2010 IEEE 3rd International Conference on Cloud Computing*, pages 51–58. IEEE, 2010.

[56] Anton Beloglazov and Rajkumar Buyya. Managing overloaded hosts for dynamic consolidation of virtual machines in cloud data centers under quality of service constraints. *IEEE Transactions on Parallel and Distributed Systems*, 24(7):1366–1379, 2013.

[57] Norman Bobroff, Andrzej Kochut, and Kirk Beaty. Dynamic placement of virtual machines for managing sla violations. In *2007 10th IFIP/IEEE International Symposium on Integrated Network Management*, pages 119–128. IEEE, 2007.

[58] Timothy Wood, Prashant Shenoy, Arun Venkataramani, and Mazin Yousif. Sandpiper: Black-box and gray-box resource management for virtual machines. *Computer Networks*, 53(17):2923–2938, 2009.

[59] Guthemberg Silvestre, Carla Sauvanaud, Mohamed Kaâniche, and Karama Kanoun. Tejo: A Supervised Anomaly Detection Scheme for NewSQL Databases. In *International Workshop on Software Engineering for Resilient Systems*, pages 114–127. Springer, 2015.

[60] IBM Knowledge Centre. "Kernel Virtual Machine (KVM): Best practices for KVM". URL `http://www.ibm.com/support/knowledgecenter/en/linuxonibm/liaat/liaatbestpractices{_}pdf.pdf`.

[61] Amin Jula, Elankovan Sundararajan, and Zalinda Othman. Cloud computing service composition: A systematic literature review. *Expert Systems with Applications*, 41(8):3809–3824, 2014.

[62] Daniel Joseph Dean, Hiep Nguyen, and Xiaohui Gu. UBL: unsupervised behavior learning for predicting performance anomalies in virtualized cloud systems. In *Proceedings of the 9th international conference on Autonomic computing*, pages 191–200. ACM, 2012.

[63] Olumuyiwa Ibidunmoye, Francisco Hernández-Rodriguez, and Erik Elmroth. Performance anomaly detection and bottleneck identification. *ACM Computing Surveys (CSUR)*, 48(1):4, 2015.

[64] Carla Sauvanaud, Guthemberg Silvestre, Mohamed Kaâniche, and Karama Kanoun. Data Stream Clustering for Online Anomaly Detection in Cloud Applications. In *Dependable Computing Conference (EDCC), 2015 Eleventh European*, pages 120–131. IEEE, 2015.

[65] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. CPI 2: CPU performance isolation for shared compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 379–391. ACM, 2013.

[66] Jian-ping Luo, Xia Li, and Min-rong Chen. Hybrid shuffled frog leaping algorithm for energy-efficient dynamic consolidation of virtual machines in cloud data centers. *Expert Systems with Applications*, 41(13):5804–5816, 2014.

[67] Jacob Leverich and Christos Kozyrakis. Reconciling high server utilization and sub-millisecond quality-of-service. In *Proceedings of the Ninth European Conference on Computer Systems*, page 4. ACM, 2014.

[68] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and QoS-aware Cluster Management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 127–144, New York, NY, USA, 2014. ACM.

[69] Luis Tomás, Carlos Vázquez, Johan Tordsson, and Ginés Moreno. Reducing Noisy-Neighbor Impact with a Fuzzy Affinity-Aware Scheduler. In *Cloud and Autonomic Computing (ICCAC), 2015 International Conference on*, pages 33–44. IEEE, 2015.

[70] Xiao-Long Liu, Xue-Bai Zhang, Hsiang Chao, and Shyan-Ming Yuan. Knee Point-Driven Bottleneck Detection Algorithm for Cloud Service System. In *Asia-Pacific Web Conference*, pages 102–111. Springer, 2016.

[71] Ravindra Singh, Bikash C Pal, and Rabih A Jabr. Statistical representation of distribution system loads using Gaussian mixture model. *IEEE Transactions on Power Systems*, 25(1):29–37, 2010.

[72] Gang Yu, Jun Sun, and Changning Li. Machine performance assessment using Gaussian mixture model (GMM). In *Systems and Control in Aerospace and Astronautics, 2008. ISSCAA 2008. 2nd International Symposium on*, pages 1–6. IEEE, 2008.

[73] Google report. `https://developers.google.com/speed/articles/web-metrics`, 2010.

[74] Jing Bi, Zhiliang Zhu, Ruixiong Tian, and Qingbo Wang. Dynamic provisioning modeling for virtualized multi-tier applications in cloud data center. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pages 370–377, July 2010.

[75] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. Cloudscale: Elastic resource scaling for multi-tenant cloud systems. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, SOCC '11, 2011.

[76] Guido Urdaneta, Guillaume Pierre, and Maarten van Steen. Wikipedia workload analysis for decentralized hosting. *Computer Networks*, 53(11), 2009. ISSN 1389-1286.

[77] Sadeka Islam, Kevin Lee, Alan Fekete, and Anna Liu. How a Consumer Can Measure Elasticity for Cloud Platforms. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, ICPE '12, pages 85–96, New York, NY, USA, 2012. ACM.

[78] D Meisner, BT Gold, and TF Wenisch. PowerNap: eliminating server idle power. *ACM SIGPLAN Notices*, 44(3):205–216, 2009.

[79] P Reviriego, V Sivaraman, Z Zhao, J A Maestro, A Vishwanath, A Sanchez-Macian, and C Russell. An energy consumption model for Energy Efficient Ethernet switches. In *High Performance Computing and Simulation (HPCS), 2012 International Conference on*, pages 98–104, 2012.

[80] K Deb, A Pratap, S Agarwal, and T Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6 (2):182–197, 2002. ISSN 1089-778X.

[81] Eckart Zitzler, Marco Laumanns, and Lothar Thiele. SPEA2: Improving the Strength Pareto Evolutionary Algorithm for Multiobjective Optimization. In K C Giannakoglou, D T Tsahalis, J Periaux, K D Papaliliou, and T Fogarty, editors, *Evolutionary Methods for Design, Optimisation and Control with Application to Industrial Problems. Proceedings of the EUROGEN2001 Conference, Athens, Greece, September 19-21, 2001*, pages 95–100, Barcelona, Spain, 2002. International Center for Numerical Methos in Engineering (CIMNE).

[82] Johannes Bader and Eckart Zitzler. Hype: An Algorithm for Fast Hypervolume-based Many-objective Optimization. *Evol. Comput.*, 19(1):45–76, 2011. ISSN 1063-6560.

[83] Jordi Guitart, Jordi Torres, and Eduard Ayguadé. A survey on performance management for internet applications. *Concurrency and Computation: Practice and Experience*, 22:68–106, 2010.

[84] Sunilkumar S Manvi and Gopal Krishna Shyam. Resource management for Infrastructure as a Service (IaaS) in cloud computing: A survey. *Journal of Network and Computer Applications*, 2013. ISSN 1084-8045.

[85] RightScale Cloud Management. `http://www.rightscale.com/`, 2012. [Online; accessed 15-May-2018].

[86] R Han, L Guo, M.M Ghanem, and Y Han, R. and Guo, L. and Ghanem, M.M. and Guo. Lightweight Resource Scaling for Cloud Applications. *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, 2012.

[87] Paweł Koperek and Włodzimierz Funika. Dynamic Business Metrics-driven Resource Provisioning in Cloud Environments. In *Parallel Processing and Applied Mathematics*, volume 7204 of *Lecture Notes in Computer Science*, pages 171–180. Springer Berlin Heidelberg, 2012.

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

[89] Emiliano Casalicchio and Luca Silvestri. Autonomic Management of Cloud-Based Systems: The Service Provider Perspective. In Erol Gelenbe and Ricardo Lent, editors, *Computer and Information Sciences III*, pages 39–47. Springer London, 2013.

[90] T C Chieu, A Mohindra, and A A Karve. Scalability and Performance of Web Applications in a Compute Cloud. In *e-Business Engineering (ICEBE), 2011 IEEE 8th International Conference on*, pages 317–323. IEEE, 2011.

[91] B Simmons, H Ghanbari, M Litoiu, and G Iszlai. Managing a SaaS application in the cloud using PaaS policy sets and a strategy-tree. In *Network and Service Management (CNSM), 2011 7th International Conference on*, pages 1–5, 2011.

[92] M Maurer, I Brandic, and R Sakellariou. Enacting slas in clouds using rules. *Euro-Par 2011 Parallel Processing*, 2011.

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

[94] Sunirmal Khatua, Anirban Ghosh, and Nandini Mukherjee. Optimizing the utilization of virtual resources in Cloud environment. In *2010 IEEE International Conference on Virtual Environments, Human-Computer Interfaces and Measurement Systems*, pages 82–87. IEEE, September 2010.

[95] RightScale. Set up Autoscaling using Voting Tags. `http://support.rightscale.com/03-Tutorials/02-AWS/02-Website_Edition/Set_up_Autoscaling_using_Voting_Tags`, 2012. [Online; accessed 15-May-2018].

[96] T C Chieu, A Mohindra, A A Karve, and A Segal. Dynamic scaling of web applications in a virtualized cloud computing environment. In *e-Business Engineering, 2009. ICEBE'09. IEEE International Conference on*, pages 281–286. Ieee, 2009.

[97] Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. Cambridge Univ Press, March 1998. ISBN 0262193981.

[98] Chris Watkins and Peter Dayan. Q-learning. *Machine learning*, 1992.

[99] Xavier Dutreilh, Sergey Kirgizov, Olga Melekhova, Jacques Malenfant, Nicolas Rivierre, and Isis Truck. Using Reinforcement Learning for Autonomic Resource Allocation in Clouds: towards a fully automated workflow. In *Seventh International Conference on Autonomic and Autonomous Systems, ICAS 2011*, pages 67–74. IEEE, May 2011.

[100] E Barrett, E Howley, and J Duggan. Applying reinforcement learning towards automating resource allocation and application scalability in the cloud. *Concurrency and Computation: Practice and Experience*, 2012.

[101] G Tesauro, N K Jong, R Das, and M N Bennani. A Hybrid Reinforcement Learning Approach to Autonomic Resource Allocation. In *Proceedings of the 2006*

IEEE International Conference on Autonomic Computing, ICAC '06, pages 65–73, Washington, DC, USA, 2006. IEEE Computer Society.

[102] Jia Rao, Xiangping Bu, Cheng Zhong Xu, Leyi Wang, and George Yin. VCONF: a reinforcement learning approach to virtual machines auto-configuration. In *Proceedings of the 6th international conference on Autonomic computing*, ICAC '09, pages 137–146, New York, NY, USA, 2009. ACM.

[103] Cheng Zhong Xu, Jia Rao, and Xiangping Bu. URL: A unified reinforcement learning approach for autonomic cloud management. *Journal of Parallel and Distributed Computing*, 72(2):95–105, February 2012.

[104] Jia Rao, Xiangping Bu, Cheng Zhong Xu, and Kun Wang. A Distributed Self-Learning Approach for Elastic Provisioning of Virtualized Cloud Resources. In *2011 IEEE 19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 45–54. IEEE, July 2011.

[105] Xiangping Bu, Jia Rao, and Cheng Zhong Xu. Coordinated Self-configuration of Virtual Machines and Appliances using A Model-free Learning Approach. *IEEE Transactions on Parallel and Distributed Systems*, pages 1–1, 2012.

[106] Qian Zhu and Gagan Agrawal. Resource Provisioning with Budget Constraints for Adaptive Applications in Cloud Environments. *IEEE Transactions on Services Computing*, 5(4):497–511, 2012.

[107] J.S. Albus. A new approach to manipulator control: The cerebellar model articulation controller (CMAC). *Transaction of the ASME, Journal of dynamic systems, measurement and control*, 1975.

[108] Daniel A. Menasce, Lawrence W. Dowdy, and Virgilio A. F. Almeida. *Performance by Design: Computer Capacity Planning By Example.* Upper Saddle River, NJ: Prentice Hall, January 2004. ISBN 0130906735.

[109] Bhuvan Urgaonkar, Prashant Shenoy, Abhishek Chandra, Pawan Goyal, and Timothy Wood. Agile dynamic provisioning of multi-tier Internet applications. *ACM Transactions on Autonomous and Adaptive Systems*, 3(1):1–39, March 2008. ISSN 15564665.

[110] D Villela, P Pradhan, and D Rubenstein. Provisioning servers in the application tier for e-commerce systems. In *Quality of Service, 2004. IWQOS 2004. Twelfth IEEE International Workshop on*, pages 57–66. IEEE, 2004.

[111] Q Zhang, L Cherkasova, and E Smirni. A regression-based analytic model for dynamic resource provisioning of multi-tier applications. In *Autonomic Computing, 2007. ICAC'07. Fourth International Conference on*, page 27. IEEE, 2007.

[112] Rui Han, Moustafa M. Ghanem, Li Guo, Yike Guo, and Michelle Osmond. Enabling cost-aware and adaptive elasticity of multi-tier cloud applications. *Future Generation Computer Systems*, June 2012.

[113] David A. Bacigalupo, Jano van Hemert, Asif Usmani, Donna N. Dillenberger, Gary B. Wills, and Stephen A. Jarvis. Resource management of enterprise cloud systems using layered queuing and historical performance models. In *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–8. IEEE, April 2010.

[114] Ahmed Ali-Eldin, Maria Kihl, Johan Tordsson, and Erik Elmroth. Efficient provisioning of bursty scientific workloads on the cloud using adaptive elasticity control. In *Proceedings of the 3rd workshop on Scientific Cloud Computing Date - ScienceCloud '12*, page 31, New York, New York, USA, June 2012. ACM Press.

[115] A Ali-Eldin, J Tordsson, and E Elmroth. An adaptive hybrid elasticity controller for cloud infrastructures. In *Network Operations and Management Symposium (NOMS), 2012 IEEE*, pages 204–212. IEEE, 2012.

[116] T Patikirikorala and A Colman. Feedback controllers in the cloud. *APSEC 2010, Cloud workshop*, 2010.

[117] Nilabja Roy, Abhishek Dubey, and Aniruddha Gokhale. Efficient Autoscaling in the Cloud Using Predictive Models for Workload Forecasting. In *2011 IEEE 4th International Conference on Cloud Computing*, pages 500–507. IEEE, July 2011.

[118] Sang Min Park and Marty Humphrey. Self-Tuning Virtual Machines for Predictable eScience. In *2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 356–363. IEEE, 2009.

[119] Harold C Lim, Shivnath Babu, Jeffrey S Chase, and Sujay S Parekh. Automated control in cloud computing: challenges and opportunities. In *Proceedings of the 1st workshop on Automated control for datacenters and clouds*, ACDC '09, pages 13–18, New York, NY, USA, 2009. ACM.

[120] Harold C. Lim, Shivnath Babu, and Jeffrey S. Chase. Automated control for elastic storage. In *Proceeding of the 7th international conference on Autonomic computing - ICAC '10*, page 1, New York, New York, USA, June 2010. ACM Press.

[121] P Padala, K Y Hou, K G Shin, X Zhu, M Uysal, Z Wang, S Singhal, and A Merchant. Automated control of multiple virtualized resources. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 13–26. ACM, 2009.

[122] Peter Bodík, Rean Griffith, Charles Sutton, Armando Fox, Michael Jordan, and David Patterson. Statistical machine learning makes automatic control practical for internet datacenters. *HotCloud'09 Proceedings of the 2009 conference on Hot topics in cloud computing*, page 12, June 2009.

[123] Alessio Gambi and Giovanni Toffetti. Modeling Cloud performance with Kriging. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 1439–1440. IEEE, June 2012.

[124] Jing Xu, Ming Zhao, Jose Fortes, Robert Carpenter, and Mazin Yousif. On the Use of Fuzzy Modeling in Virtualized Data Center Management. In *Proceedings of the Fourth International Conference on Autonomic Computing*, ICAC '07, page 25, Washington, DC, USA, 2007. IEEE Computer Society.

[125] Chengwei Wang, Krishnamurthy Viswanathan, Lakshminarayan Choudur, Vanish Talwar, Wade Satterfield, and Karsten Schwan. Statistical techniques for online anomaly detection in data centers. In *12th IFIP/IEEE International Symposium on Integrated Network Management (IM 2011) and Workshops*, pages 385–392. IEEE, 2011.

[126] Palden Lama and Xiaobo Zhou. Autonomic Provisioning with Self-Adaptive Neural Fuzzy Control for End-to-end Delay Guarantee. In *2010 IEEE International Sym-*

posium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, pages 151–160. IEEE, August 2010.

[127] E Kalyvianaki, T Charalambous, and S Hand. Self-adaptive and self-configured cpu resource provisioning for virtualized servers using kalman filters. In *Proceedings of the 6th international conference on Autonomic computing*, pages 117–126. ACM, 2009.

[128] Lixi Wang, Jing Xu, Ming Zhao, and José Fortes. Adaptive virtual resource management with fuzzy model predictive control. In *Proceedings of the 8th ACM international conference on Autonomic computing - ICAC '11*, page 191, New York, New York, USA, June 2011. ACM Press.

[129] RG Brown and RF Meyer. The fundamental theorem of exponential smoothing. *Operations Research*, 1961.

[130] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms, Chapter 32: String Matching.* McGraw-Hill Higher Education, July 2001. ISBN 0070131511.

[131] Eddy Caron, Frédéric Desprez, and Adrian Muresan. Pattern Matching Based Forecast of Non-periodic Repetitive Behavior for Cloud Clients. *Journal of Grid Computing*, 9(1):49–64, January 2011.

[132] Z Gong, X Gu, and J Wilkes. Press: Predictive elastic resource scaling for cloud systems. In *Network and Service Management (CNSM), 2010 International Conference on*, pages 9–16. IEEE, 2010.

[133] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. Cloudscale: Elastic resource scaling for multi-tenant cloud systems. *Proceedings of the 2nd ACM Symposium on Cloud Computing*, 2011.

[134] Wei Fang, ZhiHui Lu, Jie Wu, and ZhenYin Cao. RPPS: A Novel Resource Prediction and Provisioning Scheme in Cloud Data Center. In *2012 IEEE Ninth International Conference on Services Computing*, pages 609–616. IEEE, 2012.

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

[136] H Mi, H Wang, G Yin, Y Zhou, D Shi, and L Yuan. Online self-reconfiguration with performance guarantee for energy-efficient large-scale cloud computing data centers. In *Services Computing (SCC), 2010 IEEE International Conference on*, pages 514–521. IEEE, 2010.

[137] G Chen, W He, J Liu, S Nath, L Rigas, L Xiao, and F Zhao. Energy-aware server provisioning and load dispatching for connection-intensive internet services. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, volume 8, pages 337–350. USENIX Association, 2008.

[138] Sadeka Islam, Jacky Keung, Kevin Lee, and Anna Liu. Empirical prediction models for adaptive resource provisioning in the cloud. *Future Generation Computer Systems*, 28(1):155–162, 2012.

[139] Radu Prodan and Vlad Nae. Prediction-based real-time resource provisioning for massively multiplayer online games. *Future Generation Computer Systems*, 25(7): 785–793, July 2009.

[140] Sourav Dutta, Sankalp Gera, Akshat Verma, and Balaji Viswanathan. SmartScale: Automatic Application Scaling in Enterprise Clouds. In *2012 IEEE Fifth International Conference on Cloud Computing*, pages 221–228. IEEE, June 2012.

[141] Waheed Iqbal, Matthew N. Dailey, David Carrera, and Paul Janecek. Adaptive resource provisioning for read intensive multi-tier applications in the cloud. *Future Generation Computer Systems*, 27(6):871–879, June 2011.

[142] Abhishek Chandra, Weibo Gong, and Prashant Shenoy. Dynamic resource allocation for shared data centers using online measurements. *Proceedings of the 11th international conference on Quality of service*, pages 381–398, June 2003.

[143] Eddy Caron, Frédéric Desprez, and Adrian Muresan. Forecasting for Cloud computing on-demand resources based on pattern matching. Research Report RR-7217, INRIA, 2010.

[144] Tania Lorido-Botrán, José Miguel-Alonso, and Jose A. Lozano. Auto-scaling Techniques for Elastic Applications in Cloud Environments. Technical report, University of the Basque Country UPV/EHU; EHU-KAT-IK-09-12, 2012.

[145] ClarkNet HTTP Trace (From the Internet Traffic Archive). `http://ita.ee.lbl.gov/html/contrib/ClarkNet-HTTP.html`, 2012. [Online; accessed 15-May-2018].

[146] R P Prado, J Braun, J Krettek, F Hoffmann, S Garc\'\ia-Galán, J E Muñoz Expósito, and T Bertram. Gaussian Mixture Models vs. Fuzzy Rule-Based Systems for Adaptive Meta-scheduling in Grid/Cloud Computing. In *Management Intelligent Systems*, pages 295–304. Springer, 2012.

[147] Carl Edward Rasmussen. The infinite Gaussian mixture model. In *NIPS*, volume 12, pages 554–560, 1999.

[148] David M Blei, Michael I Jordan, and Others. Variational inference for Dirichlet process mixtures. *Bayesian analysis*, 1(1):121–144, 2006.

[149] Fethi Amara, Mohamed Fezari, and Hocine Bourouba. An Improved GMM-SVM System based on Distance Metric for Voice Pathology Detection. *Applied Mathematics & Information Sciences*, 10(3):1061–1070, 2016.

[150] Solomon Kullback and Richard A Leibler. On information and sufficiency. *The annals of mathematical statistics*, 22(1):79–86, 1951.

[151] A Bhattachayya. On a measure of divergence between two statistical population defined by their population distributions. *Bulletin Calcutta Mathematical Society*, 35:99–109, 1943.

[152] Keinosuke Fukunaga, editor. *Introduction to Statistical Pattern Recognition (Second Edition)*. Academic Press, Boston, second edi edition, 1990.

[153] Giorgos Sfikas, Constantinos Constantinopoulos, Aristidis Likas, and Nikolas P Galatsanos. An analytic distance metric for Gaussian mixture models with application in image retrieval. In *International Conference on Artificial Neural Networks*, pages 835–840. Springer, 2005.

[154] J K Ghosh and R V Ramamoorthi. Bayesian nonparametrics. *Springer Series in Statistics. Springer-Verlag, New York*, 16:37, 2003.

[155] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *Proceedings of the seventeenth international*

conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '12, pages 37–48, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-0759-8. doi: 10.1145/2150976.2150982.

[156] Alexander Strehl and Joydeep Ghosh. Cluster ensembles—a knowledge reuse framework for combining multiple partitions. *The Journal of Machine Learning Research*, 3:583–617, 2003.

[157] Silke Wagner and Dorothea Wagner. *Comparing clusterings: an overview.* Universität Karlsruhe, Fakultät für Informatik Karlsruhe, 2007.

[158] Marina Meila. Comparing clusterings-an information based distance. *Journal of Multivariate Analysis*, 98(5):873–895, may 2007.

[159] William M Rand. Objective criteria for the evaluation of clustering methods. *Journal of the American Statistical association*, 66(336):846–850, 1971.

[160] Eric Bae, James Bailey, and Guozhu Dong. Clustering Similarity Comparison Using Density Profiles. In *Proceedings of the 19th Australian Joint Conference on Artificial Intelligence: Advances in Artificial Intelligence*, AI'06, pages 342–351, Berlin, Heidelberg, 2006. Springer-Verlag.

[161] Lawrence Hubert and Phipps Arabie. Comparing partitions. *Journal of classification*, 2(1):193–218, 1985.

[162] Nguyen Xuan Vinh, Julien Epps, and James Bailey. Information Theoretic Measures for Clusterings Comparison: Variants, Properties, Normalization and Correction for Chance. *J. Mach. Learn. Res.*, 11:2837–2854, 2010.

[163] Danilo Horta and Ricardo J G B Campello. Comparing hard and overlapping clusterings. *Journal of Machine Learning Research*, 16:2949–2997, 2015.

[164] Jeffrey Chan, Nguyen Xuan Vinh, Wei Liu, James Bailey, Christopher A Leckie, Kotagiri Ramamohanarao, and Jian Pei. Structure-aware distance measures for comparing clusterings in graphs. In *Advances in Knowledge Discovery and Data Mining*, pages 362–373. Springer, 2014.

[165] Anne Patrikainen and Marina Meila. Comparing subspace clusterings. *IEEE Transactions on Knowledge and Data Engineering*, 18(7):902–916, 2006.

[166] Jean-Charles Lamirel. A new approach for automatizing the analysis of research topics dynamics: application to optoelectronics research. *Scientometrics*, 93(1): 151–166, 2012.

[167] Paul Jaccard. Nouvelles researches sur la distribution florale. *Bull soc vaud sci nat*, 44:223–270, 1908.

[168] Qiaoliang Xiang, Qi Mao, Kian Ming Chai, Hai Leong Chieu, Ivor Tsang, and Zhendong Zhao. A split-merge framework for comparing clusterings. *arXiv preprint arXiv:1206.6475*, 2012.

[169] Jeff M. Phillips, Parasaran Raman, and Suresh Venkatasubramanian. Generating a diverse set of high-quality clusterings. *CEUR Workshop Proceedings*, 772:80–91, 2011.

[170] Ding Zhou, Jia Li, and Hongyuan Zha. A New Mallows Distance Based Metric for Comparing Clusterings. In *Proceedings of the 22Nd International Conference on Machine Learning*, ICML '05, pages 1028–1035, New York, NY, USA, 2005. ACM.

[171] Michael H Coen, M Hidayath Ansari, and Nathanael Fillmore. Comparing Clusterings in Space. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 231–238, 2010. ISBN 9781605589077.

[172] Parasaran Raman, Jeff M Phillips, and Suresh Venkatasubramanian. Spatially-Aware Comparison and Consensus for Clusterings. In *SDM*, pages 307–318. SIAM, 2011.

[173] Eric Bae, James Bailey, and Guozhu Dong. A clustering comparison measure using density profiles and its application to the discovery of alternate clusterings. *Data Mining and Knowledge Discovery*, 21(3):427–471, 2010.

[174] Friedrich Pukelsheim. The three sigma rule. *The American Statistician*, 48(2): 88–91, 1994.

[175] Simone Romano, Nguyen Xuan Vinh, James Bailey, and Karin Verspoor. Adjusting for Chance Clustering Comparison Measures. *arXiv preprint arXiv:1512.01286*, 2015.