Multi-level monitoring and rule-based reasoning in the adaptation of time-critical cloud applications

A dissertation presented

by

Salman Taherizadeh

to

The Faculty of Computer and Information Science

in partial fulfilment of the requirements for the degree of

Doctor of Philosophy

in the subject of

Computer and Information Science

Ljubljana, 2018

APPROVAL

I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person nor material which to a substantial extent has been accepted for the award of any other degree or diploma of the university or other institute of higher learning, except where due acknowledgement has been made in the text.

— Salman Taherizadeh —

June 2018

The submission has been approved by

prof. dr. Vlado Stankovski

Associate Professor of Computer and Information Science

supervisor

University of Ljubljana

prof. dr. Denis Trček

Full Professor of Computer and Information Science

examiner

University of Ljubljana

prof. dr. Matija Marolt

Associate Professor of Computer and Information Science

examiner

University of Ljubljana

prof. dr. Radu Prodan

Associate Professor of Computer and Information Science

external examiner

University of Klagenfurt

PREVIOUS PUBLICATION

I hereby declare that the research reported herein was previously published/submitted for publication in peer reviewed journals or publicly presented at the following occasions:

[1] S. Taherizadeh and V. Stankovski. Dynamic Multi-level Auto-scaling Rules for Containerised Applications. The Computer Journal, Oxford University Press, 2018. doi: 10.1093/comjnl/bxy043.

[2] S. Taherizadeh, A.C. Jones, I. Taylor, Z. Zhao, and V. Stankovski. Monitoring self-adaptive applications within edge computing frameworks: A state-of-the-art review. Journal of Systems and Software, 136:19–38, 2018. doi: https://doi.org/10.1016/j.jss.2017.10.033.

[3] S. Taherizadeh, V. Stankovski, and M. Grobelnik. A Capillary Computing Architecture for Dynamic Internet of Things: Orchestration of Microservices from Edge Devices to Fog and Cloud Providers. The Sensors Journal, 18(9), 2018. doi: https://doi.org/10.3390/s18092938.

[4] S. Taherizadeh and V. Stankovski. Auto-scaling applications in edge computing: Taxonomy and challenges. In Proc. of the International Conference on Big Data and Internet of Thing, pages 158–163, London, United Kingdom, 2017. ACM. doi: 10.1145/3175684.3175709.

[5] S. Taherizadeh and V. Stankovski. Quality of service assurance for internet of things time-critical cloud applications: Experience with the switch and entice projects. In Proc. of 2017 6th IIAI International Congress on Advanced Applied Informatics (IIAIAAI), pages 288–293, Hamamatsu, Japan, 2017. IEEE. doi: 10.1109/IIAI-AAI.2017.209.

[6] S. Taherizadeh and V. Stankovski. Incremental learning from multi-level monitoring data and its application to component based software engineering. In Proc. of 2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC), pages 378–383, Turin, Italy, 2017. IEEE. doi: 10.1109/COMPSAC.2017.148.

[7] S. Taherizadeh, I. Taylor, A. Jones, Z. Zhao, and V. Stankovski. A network edge monitoring approach for real-time data streaming applications. In Proc. of the 13th International Conference on Economics of Grids, Clouds, Systems and Services (GECON 2016), pages 293–303, Athens, Greece, 2016. Springer. doi: https://doi.org/10.1007/978-3-319-61920-0_21.

[8] S. Taherizadeh, A. Jones, I. Taylor, Z. Zhao, P. Martin, and V. Stankovski. Runtime network-level monitoring framework in the adaptation of distributed time-critical cloud applications. In Proc. of the 22nd International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'16), pages 78-83, Las Vegas, USA, 2016. doi: 10.5281/zenodo.53869.

I certify that I have obtained a written permission from the copyright owner(s) to include the above published material(s) in my thesis. I certify that the above material describes work completed during my registration as graduate student at the University of Ljubljana.

Univerza *v Ljubljani*
Fakulteta *za računalništvo in informatiko*

Salman Taherizadeh
Večnivojski nadzor in sklepanje na podlagi pravil z namenom prilagajanja časovno-kritičnih aplikacij

POVZETEK

Računalništvo v oblaku se dandanes uporablja za postavitev različnih programskih storitev, saj omogoča najemanje računskih virov po potrebi in enostavno nadgradljivost aplikacij. Sodobni pristopi programskega inženiringa omogočajo razvoj časovno-kritičnih oblačnih aplikacij na podlagi komponent, ki so nameščeni v vsebnikih. Tehnologije vsebnikov, kot so na primer Docker, Kubernetes, CoreOS, Swarm, OpenShift Origin in podobno, omogočajo razvoj zelo dinamičnih oblačnih aplikacij, pod pogojih stalno spreminjajočih obremenitev. Oblačne aplikacije, ki temeljijo na tehnologijah vsebnikov zahtevajo prefinjene metode samodejnega prilagajanja, z namenom delovanja pod različnimi pogoji delovnih obremenitev, na primer pod pogojih drastičnih sprememb delovnih obremenitev.

Predstavljajmo si socialno omrežje, ki je oblačna aplikacija in v katerem se določena novica začne bliskovito širiti. Po eni strani potrebuje oblačna aplikacija zadosti računskih virov, še pred nastankom delovne obremenitve. Po drugi strani je najem dragih oblačnih infrastruktur v daljšem časovnem obdobju nepotreben in zato tudi nezaželen. Izbira metode samodejnega prilagajanja oblačne aplikacije tako pomembno vpliva na parametre kakovosti storitve, kot sta odzivni čas in stopnja uporabe računskih virov. Obstoječi sistemi za orkestracijo vsebnikov, kot sta npr. Kubernetes in sistem Amazon EC2, uporabljajo avtomatska pravila s statično določenimi pragovi, ki se zanašajo predvsem na infrastrukturne metrike, kot sta na primer uporaba procesorja in pomnilnika.

V tej doktorski disertaciji predstavljamo novo metodo dinamičnega večstopenjskega (angl. Dynamic Multi-Level, DM) samodejnega prilagajanja oblačnih aplikcij, ki uporablja poleg infrastrukturnih metrik tudi aplikacijske metrike s spreminjajočimi se pragovi. Novo DM metodo smo vgradili v delujočo arhitekturo sistema za samoprilagajanje aplikacij. Novo metodo DM primerjamo s sedmimi obstoječimi metodami samodejnega prilagajanja pri različnih scenarijih sintetičnih in realne delovne obremenitve. Primerljivi pristopi samodejnega prilagajanja vključujejo metode Kubernetes Horizontal Pod Autoscaling (HPA), Step Scaling 1 in 2 (SS1, SS2), Target Tracking Scaling 1 in 2 (TTS1, TTS2) ter Static Threshold Based Scaling 1 in 2 (THRES1, THRES2). Vse obravnavane metode samodejnega prilagajanja trenutno štejejo kot zelo napredni pristopi, ki se uporabljajo v proizvodnih sistemih, kot so sistemi temelječi na tehnologijah Kubernetes in Amazon EC2. Scenariji delovnih obremenitev, ki jih uporabljamo v tem delu predstavljajo vzorce vztrajno naraščajočih/padajočih, drastično spreminjajočih, rahlih sprememb ter dejanskih delovnih obremenitev.

Na podlagi rezultatov poskusov, opravljenih za vsak vzorec delovnih obremenitev posebej, smo primerjali vseh osem izbranih metod samodejnega prilagajanja glede na odzivni čas in število instanciranih vsebnikov. Rezultati kot celota kažejo, da ima predlagana nova metoda DM večjo splošno samoprilagodljivost v primerjavi s preostalimi metodami. Zaradi zadovoljivih rezultatov smo predlagano metodo DM vgradili v sistem SWITCH za programski inženiring časovno-kritičnih oblačnih aplikacij. Pravila za samoprilagajanje aplikacij in druge informacije, kot so na primer lastnosti platform za virtualizacijo, trenutne obremenitve aplikacije, ponavljajoče se zahteve po višji kakovosti storitev in podobno, se nenehno shranjujejo v obliki Resource Description Framework (RDF) trojk v bazi znanja, ki je tudi vključena v predlagani arhitekturi. Ključna zahteva za razvoj baze znanja, je omogočiti vsem deležnikom programske platforme SWITCH, kot so na primer ponudniki oblačnih storitev, možnost integracije informacij, analizo daljših trendov in podporo strateškemu planiranju.

Ključne besede: časovno-kritične aplikacije, računalništvo v oblaku, samoprilagajanje, dinamični pragovi, večnivojski nadzor, virtualizacija z uporabo vsebnikov

University *of Ljubljana*
Faculty *of Computer and Information Science*

Salman Taherizadeh
Multi-level monitoring and rule-based reasoning in the adaptation of time-critical cloud applications

ABSTRACT

Nowadays, different types of online services are often deployed and operated on the cloud since it offers a convenient on-demand model for renting resources and easy-to-use elastic infrastructures. Moreover, the modern software engineering discipline provides means to design time-critical services based on a set of components running in containers. Container technologies, such as Docker, Kubernetes, CoreOS, Swarm, OpenShift Origin, etc. are enablers of highly dynamic cloud-based services capable to address continuously varying workloads. Due to their lightweight nature, they can be instantiated, terminated and managed very dynamically. Container-based cloud applications require sophisticated auto-scaling methods in order to operate under different workload conditions, such as drastically changing workload scenarios.

Imagine a cloud-based social media network website in which a piece of news suddenly becomes viral. On the one hand, in order to ensure the users' experience, it is necessary to allocate enough computational resources before the workload intensity surges at runtime. On the other hand, renting expensive cloud-based resources can be unaffordable over a prolonged period of time. Therefore, the choice of an auto-scaling method may significantly affect important service quality parameters, such as response time and resource utilisation. Current cloud providers, such as Amazon EC2 and container orchestration systems, such as Kubernetes employ auto-scaling rules with static thresholds and rely mainly on infrastructure-related monitoring data, such as CPU and memory utilisation.

This thesis presents a new Dynamic Multi-Level (DM) auto-scaling method with dynamically changing thresholds used in auto-scaling rules which exploit not only infrastructure, but also application-level monitoring data. The new DM method is implemented to be employed according to our proposed innovative viable architecture for auto-scaling containerised applications. The new DM method is compared with seven existing auto-scaling methods in different synthetic and real-world workload scenarios. These auto-scaling approaches include Kubernetes Horizontal Pod Auto-scaling (HPA), 1st method of Step Scaling (SS1), 2nd method of Step Scaling (SS2), 1st method of Target Tracking Scaling (TTS1), 2nd method of Target Tracking Scaling (TTS2), 1st method of static THRESHOLD-based scaling (THRES1), and 2nd method of static Threshold-based scaling (THRES2). All investigated auto-scaling methods are currently considered as advanced approaches, which are used in production systems such as Kubernetes, Amazon EC2, etc. Workload scenarios which are examined in this work also consist of slowly rising/falling workload pattern, drastically changing workload pattern, on-off workload pattern, gently shaking workload pattern, and real-world workload pattern.

Based on experimental results achieved for each workload pattern, all eight auto-scaling methods are compared according to the response time and the number of instantiated containers. The results as a whole show that the proposed DM method has better overall performance under varied amount of workloads than the other auto-scaling methods. Due to satisfactory results, the proposed DM method is implemented in the SWITCH software engineering system for time-critical cloud-based applications. Auto-scaling rules along with other properties, such as characteristics of virtualisation platforms, current workload, periodic QoS fluctuations and similar, are continuously stored as Resource Description Framework (RDF) triples in a Knowledge Base (KB), which is included in the proposed architecture. The primary reason to maintain the KB is to address different requirements of the SWITCH solution stakeholders, such as those of cloud-based service providers, allowing for seamless information integration, which can be used for long-term trends analysis and support to strategic planning.

Key words: time-critical applications, cloud computing, self-adaptation, dynamic thresholds, multi-level monitoring, container based virtualisation

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my supervisor, Assoc. Prof. Vlado Stankovski, who has provided me with invaluable guidance and advice throughout my PhD candidature. I wish to express my sincere gratitude to him without whose support this dissertation would not have been possible.

I am grateful to Prof. Marko Grobelnik at the Jožef Stefan Institute for his support, and Dr. Danijel Skočaj, Dr. Tomaž Curk, and Dr. Zoran Bosnić for their continuous efforts throughout the PhD programme at the Faculty of Computer and Information Science, University of Ljubljana.

I would like to thank all my colleagues at the SWITCH, ENTICE, DECENTER and PrEstoCloud projects for their co-operation in doing my research work without interruption.

Finally, my appreciation also goes to my family and friends for their encouragement throughout the period during which I worked on this thesis.

— Salman Taherizadeh, Ljubljana, June 2018.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Introduction

In recent years, a wide variety of time-critical software systems, such as Internet of Things (IoT) solutions (e.g. vehicle tracking) and data analytics applications (e.g. finite element analysis), for which the application response time is important, have emerged as cloud-based services. This is because scalability is the key enabler of the cloud usage. It means that cloud computing is capable of offering dynamic on-demand scalable resource allocation to provide favourable Quality of Service (QoS) for users. It is able to dynamically provision and de-provision cloud-based resources in response to the current demand in real-time.

The change in the workload demands of cloud-based applications may happen in different ways. For example, a broadcasting news channel unexpectedly receives a heavy workload since a video or some news suddenly spreads in the social media world. Another example is a cloud-based batch processing system for which requests tend to be accumulated around batch runs regularly over only short periods of time. These types of systems generally have short active periods, between which the application can be provided at the lowest service level.

Concrete examples of scalable cloud-based infrastructure providers are Amazon EC2 [1] and Microsoft Azure [2], which can manage situations if resources allocated to an application should be dynamically increased or decreased when the workload varies over time. However, scalability of such cloud-based infrastructures is not a panacea to achieve high operational application performance. If auto-scaling technologies offered by cloud-based infrastructure providers cannot sufficiently provision the resources needed, application quality will be inappropriate and hence users will be turned away. Moreover if they provision the resources more than what would be needed, the potential cost-saving advantage of using the cloud will be compromised.

1.1 Research problems

The intention behind this work is to address the shortcomings of current auto-scaling methods within cloud computing frameworks, as shown in Fig. 1.1.

Providing high-quality results under the conditions of dynamically changing workload intensity is necessary for time-critical cloud-based applications in order to make them practical in a business context. Although, reactive auto-scaling methods as the focus of this thesis employ rules with fixed thresholds which are mainly based on infrastructure-level parameters such as Central Processing Unit (CPU) and memory utilisation. This includes reactive auto-scaling mechanisms used by commercial Virtual Machine (VM)-

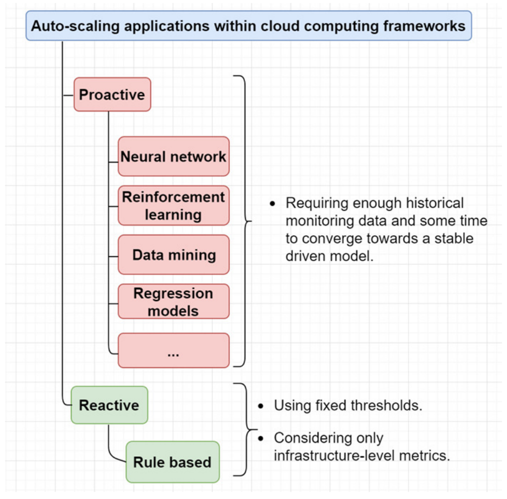

Auto-scaling applications within cloud computing frameworks
Proactive
Neural network
Reinforcement learning
Data mining
Regression models
...
Requiring enough historical monitoring data and some time to converge towards a stable driven model.
Reactive
Rule based
Using fixed thresholds.
Considering only infrastructure-level metrics.

Figure 1.1

Summary of the research problems.

based cloud infrastructure providers such as Amazon EC2 [1] and Microsoft Azure [2], and also open-source container orchestrators such as Kubernetes [3] and OpenShift Origin [4]. Such rule-based auto-scaling methods may be useful for some basic types of cloud-based applications. However, their resource utilisation and performance drops when time-critical applications need to be used [5]. These static rule-based auto-scaling mechanisms are not flexible enough to adjust themselves to the status of the execution environment at runtime.

As an example, a CPU-based auto-scaling rule can be specified in a way that more VMs/containers will be instantiated if the average utilisation of CPU reaches a fixed threshold such as 80%; while some VMs/containers may be stopped if the average CPU usage is less than 80%. Such settings cannot be very helpful for specific workload scenarios e.g. drastically changing workload patterns. Furthermore, these auto-scaling rules cause a stable system at 80% resource utilisation at the best case, which means 20% of resources are wasted, that is not profitable. One of the significant challenges and main technical issues in providing an auto-scaling method is to define to what extent the auto-scaling mechanism can be self-adjustable to runtime variations in running conditions.

Moreover, proactive auto-scaling methods [6–9] are developed to predict the amount of resources required in the near future based on collected historical monitoring data, current intensity of workload, QoS of the application, etc. Proactive auto-scaling approaches use learning algorithms such as reinforcement learning [10, 11], neural network

[12, 13], queuing theory [14, 15], data mining [16, 17] and regression models [18, 19] to anticipate the amount of required resources. It should be noted that these methods require enough historical data to train a performance model and some time to converge towards a stable driven model. Therefore, if proactive auto-scaling methods have a large enough training data set reflecting characteristics of all different possible operational situations, they are capable of generalising that means they can react to unseen changing workload scenarios. As a consequence, if the training data set is not comprehensive enough, such proactive approaches may suffer from their imprecision limit which may result in whether over-provisioning problem or serious performance drops.

1.2 Research objectives

Auto-scaling methodologies used by time-critical cloud-based applications still can be significantly improved [20–25]. An auto-scaling method which is unable to address changing workload intensity over time will result in either resource over-provisioning situation where the usage of allocated resources is unacceptably low or resource under-provisioning situation where the application suffers from poor performance. Therefore, a fine-grained auto-scaling method is required in response to dynamic fluctuations in workload at runtime.

During the execution time, a fine-grained auto-scaling method dynamically allocates the optimal amount of resources required to support application performance with neither resource under-provisioning nor over-provisioning. Such an auto-scaling approach should be capable of satisfying application performance requirements (such as response time constraints), while optimising the utilisation of resources allocated to the application (such as number of VMs/containers), as shown in Fig. 1.2.

Therefore, below two objectives which our proposed auto-scaling method should fulfil are outlined:

- *Improving application performance of time-critical cloud-based applications*: Adapting time-critical cloud-based applications to the changing execution environment such as workload variations at runtime is essential in order to ensure application performance requirements in terms of response time. Cloud-based applications need auto-scaling methods which are able to meet service response time constraints under different workload conditions.

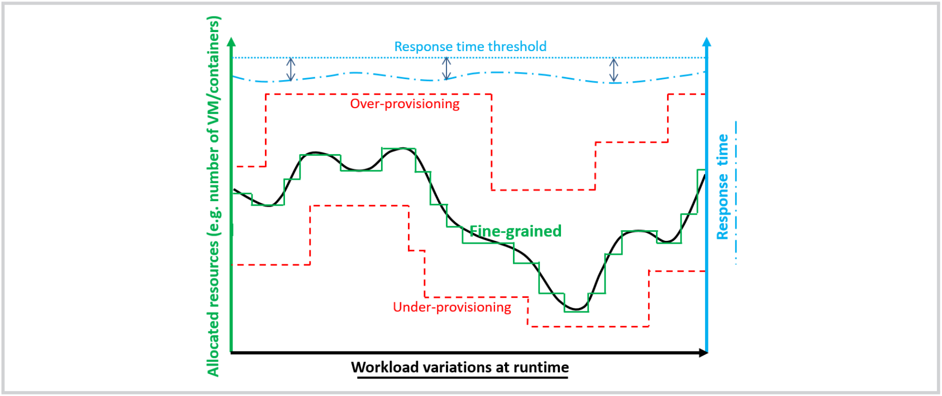


Figure 1.2

Fine-grained auto-scaling during workload variations at runtime.

- *Optimising resource utilisation in the cloud*: An auto-scaling mechanism should continuously adjust the optimal amount of resources needed to address the performance objective of time-critical cloud-based applications. Releasing of idle resources when they are not used anymore is advantageous for better resource utilisation in the cloud. Low resource utilisation is still a serious problem in cloud datacentres [26–29]. Resource utilisation should be maximised in order to save resources in the cloud environment and reduce energy consumption in cloud computing.

1.3 Contributions to science

The key contribution of this thesis can be summarised as follows:

- *presenting a new multi-level monitoring framework to address the whole spectrum of monitoring requirements for auto-scaling containerised time-critical applications to be published under the Apache 2 license*: One of the main contributions of this research work is to address needs of containerised self-adaptive time-critical cloud-based applications, with a newly designed monitoring approach, addressing the complete life-cycle of time-critical services, starting from their design, engineering, deployment and operational stages. The monitoring framework has to address various concerns, such as those of the end-users, the application and datacentre operators. Therefore, a new multi-level monitoring framework is needed, which would elaborate the needs and relationships between various QoS at these levels: infrastructure-level, container-level and application-level. Over the recent years, a large number of widely used cloud monitoring tools have been provided and applied in both the industry and academic research. However, none of these monitoring tools address the complete software engineering life-cycle of adaptive cloud-based applications.

- *proposing innovative dynamic thresholds employed for upscaling and downscaling actions in response to the changing workload over time*: In recent years, many of research works have provided different rule-based auto-scaling methods which use only static, fixed thresholds. Although such rule-based methods may be useful for some basic types of workloads, their performance and resource utilisation drops in an environment with special workload patterns such as drastically changing scenarios. One of the main open challenges in proposing a rule-based auto-scaling technique is to decide to what extent the adaptation approach should be self-adjustable to changes in the execution environment. To come up with this challenge, in our method, the focus is to extend this area of research by presenting auto-scaling rules that apply dynamic thresholds instead of fixed scaling rules.
- *introducing a fine-grained auto-scaling method able to not only meet favourable service quality, but also provide optimal resource utilisation*: Fine-grained auto-scaling mechanisms are needed to cope with highly dynamic workloads in the cloud environment. Existing traditional application adaptation approaches unfortunately cannot accurately provide favourable service quality while offering optimal resource utilisation. This thesis introduces a new auto-scaling method which applies multi-level monitoring data used in dynamic rules to automatically increase or decrease the total number of computing instances in order to accommodate varied workloads, while producing acceptable response time.

1.4 Thesis structure

The rest of this thesis is organized as follows. Chapter 2 describes the background relevant to container-based virtualisation and multi-level monitoring. Chapter 3 presents the state-of-the-art focusing on the auto-scaling of applications from viewpoints of both experience studies and existing production solutions. Chapter 4 explains the architecture of our proposed auto-scaling method in detail, while empirical results are presented in Chapter 5. Chapter 6 contains a critical discussion of the proposed auto-scaling method, which is followed by conclusions in Chapter 7.

2

Background

In this thesis, container-based virtualisation technology is used to deploy and run time-critical cloud-based applications. In comparison to VMs, containers co-located on the same host share the kernel of the host Operating System (OS), and hence each container does not include an entire operating system. Because of this distinction, container-based virtualisation is an alternative to VM-based virtualisation in order to wrap cloud-based service instances into lightweight, portable containers. This fact makes this new technology an appropriate way to build scalable, rapidly deployable services in the cloud. It means that containers can be quickly launched or terminated when necessary since they do not need to start or stop operating system that can take significant amount of time.

If container-based virtualisation is used to scale services in the cloud, container-level monitoring becomes mandatory, in addition to the application-level monitoring. Therefore, it can be concluded that employing a multi-level monitoring system able to address the whole spectrum of monitoring requirements is necessary for auto-scaling containerised time-critical applications.

This chapter has been divided into two sections in order to describe the adoption of container-based virtualisation and the necessity of using multi-level monitoring respectively.

2.1 Container-based virtualisation

Hypervisor-based virtualisation technologies are able to support standalone VMs which are independent and isolated of the host machine. Each VM instance has its own operating system and a set of libraries, and operates within an emulated environment provided by the hypervisor. If the system uses VMs to run application services, VM-level monitoring becomes mandatory. However nowadays, a modern software engineering discipline provides an approach to design cloud-based applications based on a set of components running in containers [30] rather than VMs, shown in Fig. 2.1.

Different from VMs, the utilisation of containers does not need an operating system to boot up that has gained increasing popularity in the cloud computing frameworks. Resource usage of VMs is extensive and thus typically they cannot be easily developed on small servers or resource-constrained devices, in contrast to containers. Tab. 2.1 provides a comparison between container-based and VM-based virtualisation [31].

Since their nature is lightweight, deployment of containerised time-critical services at runtime can be accomplished faster than VMs, and hence they have the ability to allow for quick scaling of cloud-based applications. The lightweight nature and portability of

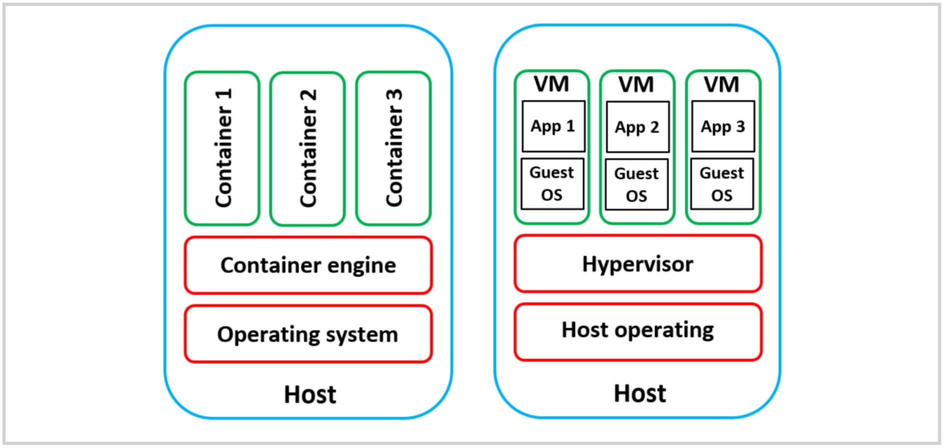


Figure 2.1

Container versus VM-based virtualisation.

Table 2.1

Container-based vs VM-based virtualisation

Feature	Containers	VMs
Requirement	Container engine e.g. Docker [32]	Hypervisor e.g. Xvisor [33]
Weight	Lightweight	Heavyweight
Boot time	Fast	Slow
Footprint	Smaller	Bigger

containers make it easy to dynamically handle changing workloads, scaling up or scaling down applications when the workload varies over time. In this way, various container-based virtualisation platforms such as Google Container Engine (GKE) [34] and Amazon EC2 Container Service (ECS) [35] have been offered as alternatives to hypervisor-based virtualisation.

Experiments in this thesis are based on Docker which is a container technology for Linux that allows a developer to package up an application with all of the parts needed [36]. Among all container virtualisation platforms, Docker has been recently very popular and rapidly developing [37]. Docker offers a set of APIs exposed by the Docker engine for creating, instantiating, terminating and managing containers, and also building images, sharing them through repositories and looking for images created and made publically available by any other developer. Docker Engine is the core of Docker which provides APIs for starting, stopping, resuming and removing containers.

When a container instance is started, there is a possibility to specify how the container interacts with the host system. To this end, ports can be configured by using a port mapping between internal and external port numbers. In this way, other hosts whether on the same network or on the Internet can communicate with the container via the external port, and the connection will be mapped to the internal port inside the container.

Size of container images is significantly smaller than the size of full VM images. There is a repository named Docker Hub repository [38] by which all developers can share their public container images provided for different purposes, shown in Fig. 2.2. The Docker Hub repository is a database for storing and distributing Docker images. The Docker hub repository exposes APIs which can be called to push (store) and pull (retrieve) container images [39]. The Docker registry can be also installed locally to store Docker images. When a local registry is used, pulling container images will be faster, and hence running container instances will be quick. Using a local Docker registry significantly decreases deployment latency as well as network overhead across the network.

All Docker images stored in this registry are identified by their name, while different versions of a container image can be stored under the same name. A Docker image is generally described in a text file called Dockerfile [40]. Dockerfile consists of consecutive steps on how to create a specific Docker image, what base image should be used, which dependencies need to be downloaded, what commands have to run to execute the containerised application, port numbers which the container listens for network connections at runtime, and so on. In this thesis, all developed container images used for experiments along with their associated Dockerfiles are publically released under an Apache 2 license [41].

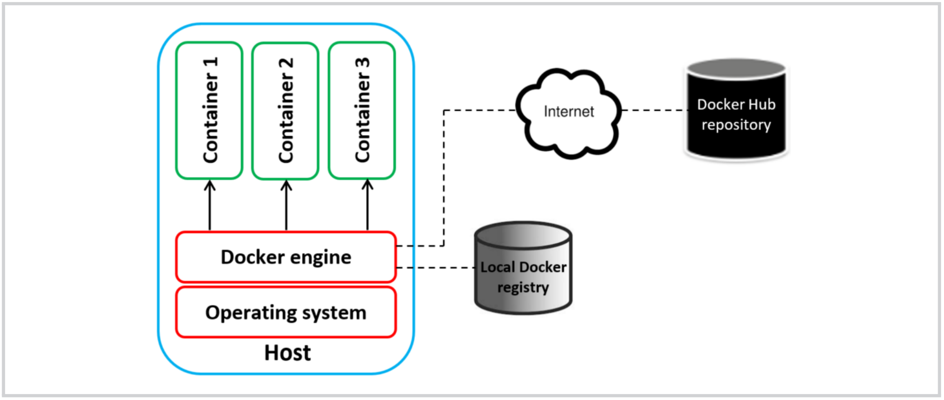


Figure 2.2

Docker Hub repository.

2.2 *Multi-level monitoring*

The significance of a monitoring system fully-fitted for auto-scaling cloud-based applications has been presented in many academic and commercial-purposed solutions [42–52] in various contexts. The monitoring system should be able to address the whole spectrum of requirements which are needed for cloud-based applications within auto-scaling frameworks. These requirements for monitoring systems within auto-scaling cloud-based frameworks are explained as follows:

- *Open Source*: An open source monitoring system is software with source code which can be shared, inspected, modified and enhanced by anyone because its design is publicly accessible. This requirement is a great opportunity for easier customisation and integration possibilities into other systems.

- *License*: Open-source licenses affect the way how other developers can study, use, modify and distribute software. The Apache 2 license is an open-source software license released by the Apache Software Foundation (ASF) [53]. The utility of this license is motivated in research works since it is a popular and widely deployed license supported by a strong community.

- *Scalability*: A monitoring system should be scalable which means that it is able handle a notable number of monitored entities such as resources and services [54]. This monitoring characteristic is a very important requirement in auto-scaling cloud-based applications due to the necessity of considering a wide variety of parameters to be monitored across various levels of framework.

- *Alerting*: In cloud-based computing scenarios especially for auto-scaling applications, there is a significant requirement for monitoring systems allowing us to define custom alert rules which meet particular criteria. For instance, a monitoring system should be capable of triggering notifications if a given container or VM instance starts behaving irregularly or if a metric violates its associated threshold.

- *Time Series Database (TSDB)*: This requirement is a significant competency to developing a monitoring system which needs to be optimised for the long-term storage and retrieval of monitoring data. Therefore, the monitoring data can be employed to specify future auto-scaling strategies.

- *Graphical User Interface (GUI)*: A monitoring solution used for auto-scaling cloud-based applications should be able to provide GUI as well as APIs to display and expose runtime monitoring statistics about resources and services being monitored. The GUI and remote APIs need to be accessible by other entities.

To adapt containerised applications to the changing workload at runtime and satisfy application QoS, it is essential to apply a comprehensive monitoring system capable of considering different monitoring levels including: (i) container-level monitoring and (ii) application-level monitoring, described in the next two subsections.

2.2.1 *Container-level monitoring*

If containers are used to run time-critical applications, container-level monitoring becomes mandatory when designing an auto-scaling mechanism. A container-level monitoring system should be able to monitor a container's key attributes and display their runtime values, such as CPU, memory, storage and network traffic usage of the container instance. Tab. 2.2 shows a list of tools usable specifically for the purpose of monitoring containers and exposing runtime value of their associated metrics.

Table 2.2

Overview of container-level monitoring tools

Tool	Open Source	License	Scalability	Alerting	TSDB	GUI
cAdvisor	Yes	Apache 2	No	No	No	Yes
cAdvisor+InfluxDB+Grafana	Yes	Mixed[a]	Yes	No	Yes	Yes
Prometheus	Yes	Apache 2	No	Yes	Yes	Yes
DUCP	Yes	Commercial	Yes	Yes	No	Yes
Scout	Yes	Commercial	No	Yes	Yes	Yes

[a]Using three tools together: cAdvisor (Apache 2) + InfluxDB (MIT) + Grafana (Apache 2)

cAdvisor [55] is a system specifically designed for measuring, aggregating and showing monitoring data about running containers. It can be also employed to provide instant visibility across all containers managed by any container orchestration system such as Kubernetes [3], ECS [35], Mesos [56] and Docker Swarm [57]. cAdvisor runs in a container and employs the Docker Remote API [58] to obtain the statistics associated to containers, shown in Fig. 2.3.

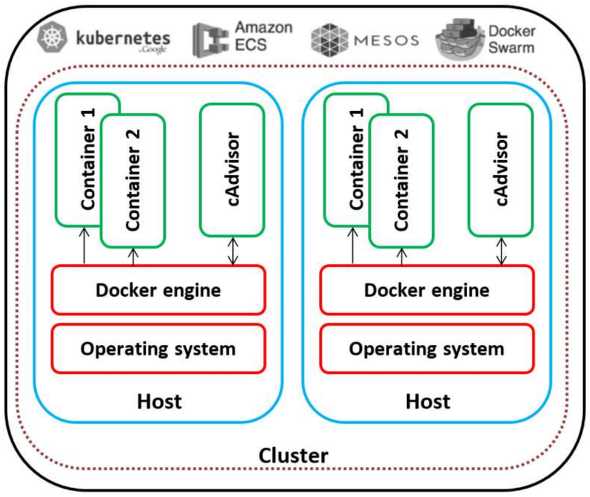


Figure 2.3

cAdvisor monitoring system.

Therefore, starting this monitoring system is easy since it is shipped in a single container. Once cAdvisor is launched, it will hook itself into the Docker daemon operating on the host machine and start collecting metrics from all running containers, including the cAdvisor container instance itself. In order to use the cAdvisor Web-based interface, the URL http://localhost:8080/ needs to be opened in a browser.

A minute's worth of monitoring data can be used for awareness of real-time metrics about running containers over time. cAdvisor only displays monitoring information measured during the last 60 seconds. It means that it is not possible to view any features further back with using only a standard installation of cAdvisor. However, cAdvisor can store the monitoring data in an external TSDB such as InfluxDB [59], Elasticsearch [60] or BigQuery [61] which allows long-term storage and analysis of measured values. Moreover, Grafana [62] can be applied as Web-based interface to visualise time-series monitoring data. In this way, one cAdvisor container is responsible for data collection on each host, and then it sends the collected monitoring data to the TSDB for long-term storage, and afterwards Grafana provides a custom display panel which is a graphical display dashboard. This architecture is shown in Fig. 2.4.

Since the cAdvisor monitoring system is limited to the latest one-minute data of a single host, use of InfluxDB and Grafana on top of cAdvisor significantly improves scalability of the monitoring system and also visualising the monitored data in informative charts and statistics for any time periods.

Prometheus [63] is an open-source monitoring solution which includes a TSDB. It is able to gather monitoring metrics at various time intervals, and show the monitoring measurements. The Prometheus alert-manager investigates alert conditions, and triggers notifications when the system starts experiencing abnormal situation. However cAdvisor is the easier monitoring system to be employed for monitoring containers in comparison to Prometheus, it has restrictions with alert management. It should be noted that both may not be able to appropriately offer turnkey scalability to handle large number of containers. Prometheus can be also used with cAdvisor and Grafana. In this way, the monitoring data collected by cAdvisor can be stored in a Prometheus database, and then can be visualised via a Grafana dashboard, shown in Fig. 2.5.

Use of Prometheus, instead of InfluxDB can be preferred because it stores a name and additional labels for each monitoring metric effectively only once, whereas InfluxDB redundantly stores them for every timestamp [64].

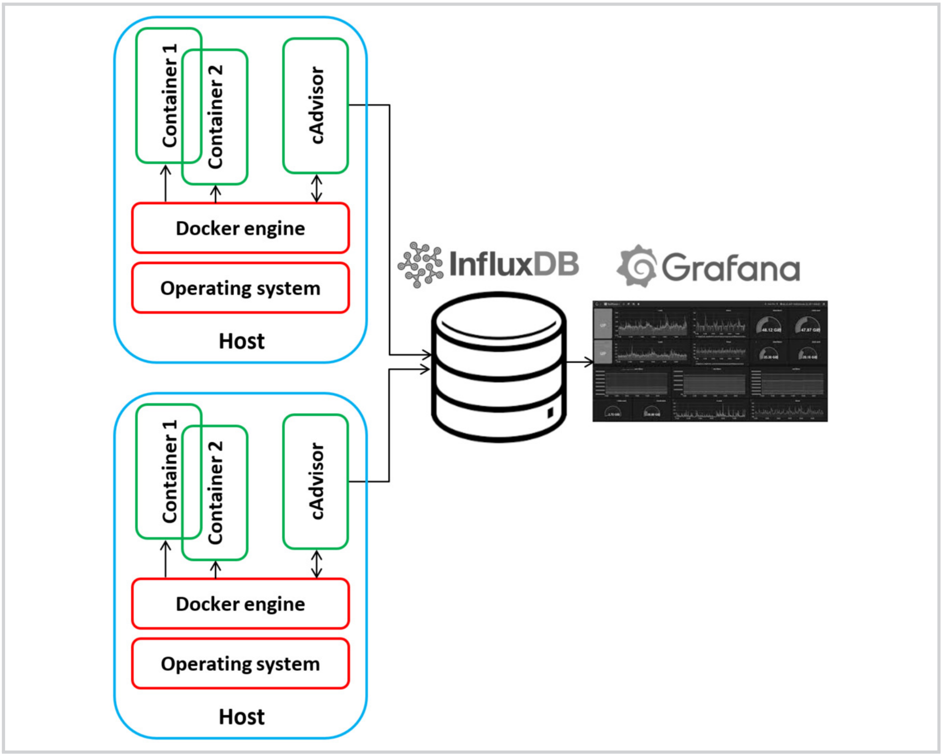


Figure 2.4

Using InfluxDB and Grafana on top of cAdvisor.

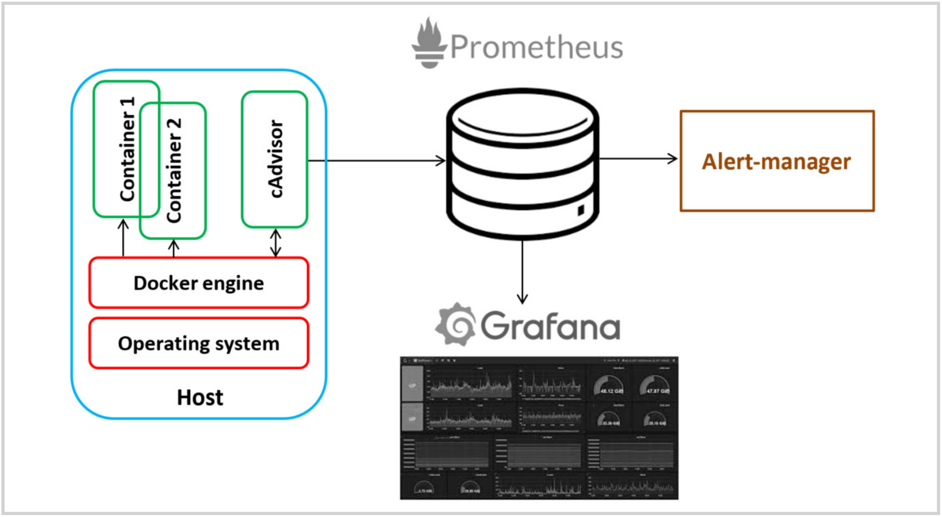


Figure 2.5

Using Prometheus and Grafana on top of cAdvisor.

Docker Universal Control Plane (DUCP) [65] is a commercial tool to monitor, deploy, configure and manage Docker containerised applications through a single graphical interface. High scalability and Web-based user interface are the substantial characteristics of this container management solution. Scout [66] is also a commercial container monitoring solution which includes a Web-based graphical interface management dashboard. It is capable of storing measured monitoring values taken from containers during maximum 30 days. This monitoring solution also supports notification and alerting of events specified by predetermined thresholds.

2.2.2 *Application-level monitoring*

An application-level monitoring system should be able to check the current status of the application performance (such as response time or application throughput) and present the monitoring information over the execution time [67]. Application-level monitoring is essential to achieve a high-level reliability and availability of cloud-based applications, and it is required for auto-scaling operations in terms of adaptive services to the varying workload.

However there are many monitoring systems able to measure the situation of the underlying cloud-based infrastructures, monitoring application-specific metrics is still a challenge which needs to be improved by both academic and industry cloud communities. To this end, the monitoring system should be extensible enough since each application has its own definition of QoS. In this way, cloud-based application providers may be able to provide a monitoring solution specifically prepared for their own services.

Tab. 2.3 presents a number of well-known monitoring platforms which are able to measure cloud-based application-level metrics, in addition to infrastructure-level parameters.

Zenoss [68, 69] monitoring platform relies on agent-less data collection mainly based on the Simple Network Management Protocol (SNMP) protocol. This monitoring solution has an open architecture which allows customers to appropriately customise it according to their monitoring needs. Zenoss is able to provide comprehensive real-time models for managed applications as well as other resources such as VMs or networks. It presents holistic performance and health insights. Zenoss also provides event management and reporting. However, it has a limited open-source version and the full version for this monitoring platform requires payment, hence its utility in research works would be declined. The raw monitoring data is collected by collectors and then forwarded to the Zenoss Core. This monitoring system supports as many entities (e.g. applications, VMs, etc.) as customers choose to monitor. Currently, the largest deployment includes over 32,000 entities being monitored. The basic architecture of Zenoss monitoring platform is shown in Fig. 2.6.

Ganglia [70, 71] is a scalable monitoring system generally for high-performance computing environments such as clusters and grids. This monitoring tool has been recently extended to monitor private and public clouds (e.g. via sFlow [72]). Moreover, cloud-based solutions such as Cassandra TSDB usable in many scaling scenarios are integrated with Ganglia. However this tool is typically designed to monitor infrastructure-level

Table 2.3

Overview of application-level monitoring tools

Tool	Open Source	License	Scalability	Alerting	TSDB	GUI
Zenoss	Yes	GPL	Yes	Yes	Yes	Yes
Ganglia	Yes	BSD	Yes	No	Yes	Yes
Zabbix	Yes	GPL	Yes	Yes	Yes	Yes
Lattice	Yes	Apache 2	Yes	No	No	No
JCatascopia	Yes	Apache 2	Yes	No	Yes	Yes

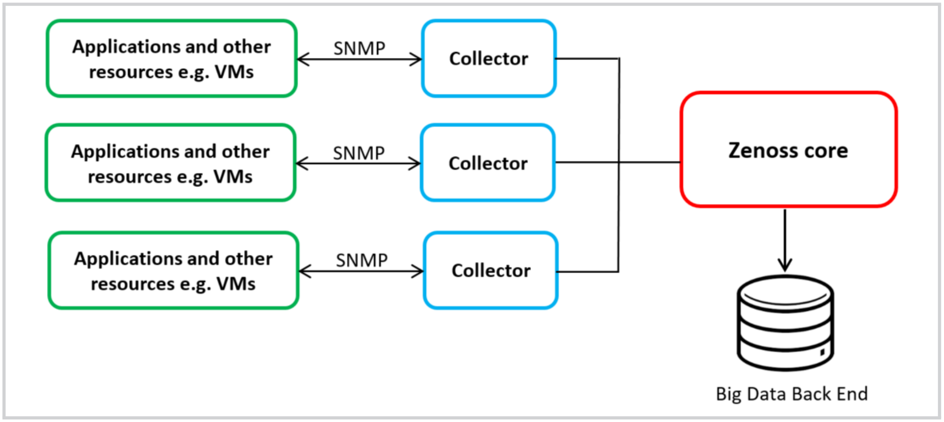

Applications and other resources e.g. VMs
SNMP
Collector
Applications and other resources e.g. VMs
SNMP
Collector
Applications and other resources e.g. VMs
SNMP
Collector
Zenoss core
Big Data Back End

Figure 2.6

Zenoss monitoring solution's architecture.

metrics (such as CPU utilisation, load average, disk free, and so on) about machines in clusters and display both runtime and historical monitoring data as a set of graphs in a Web-based graphical interface, Ganglia's metric libraries can be extended to monitor application-level parameters. It applies widely used protocols such as External Data Representation (XDR) and Extensible Markup Language (XML) to collect monitoring states, and also it uses RRDtool [73] to store and visualise the time series data. Ganglia consists of two main parts: Gmond and Gmetad. The Gmond is a daemon which sits on every single node being monitored, and gathers monitoring statistics. The Gmetad is a daemon which periodically polls Gmonds and stores their monitoring metrics into an RRDtool database. The basic architecture of Ganglia monitoring system is shown in Fig. 2.7.

Zabbix [74, 75] is designed to be an open-source agent-based monitoring system. The Zabbix server runs on a standalone machine which collects and aggregates monitoring data sent by the Zabbix agents. The Zabbix monitoring system supports the alerting feature to trigger notification if any predefined situation occurs, such as when the CPU utilisation is over 80%. Zabbix is mainly designed to monitor network parameters and network services. SQL database is used to store monitoring data, and a Web-based front-end and an API are provided to access the measured values. The native Zabbix agent is implemented in the C language, and hence it has a relatively small footprint. However, the auto-discovery feature of Zabbix may be inefficient [76]. This is because sometimes it may take around five minutes to discover that a monitored node is no longer running in the environment. This limitation can be a significant problem for any cloud-based

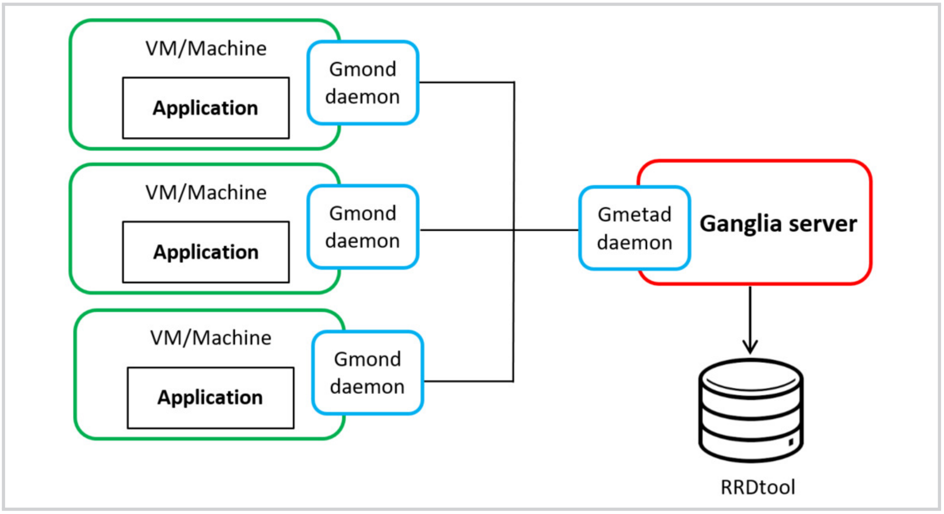


Figure 2.7

Ganglia architecture.

time-critical auto-scaling scenario. The Zabbix monitoring servers can be hierarchically arranged in a way that child Zabbix servers forward all monitoring data to a parent Zabbix server which stores the data in a database. The Zabbix monitoring system's hierarchical architecture is shown in Fig. 2.8.

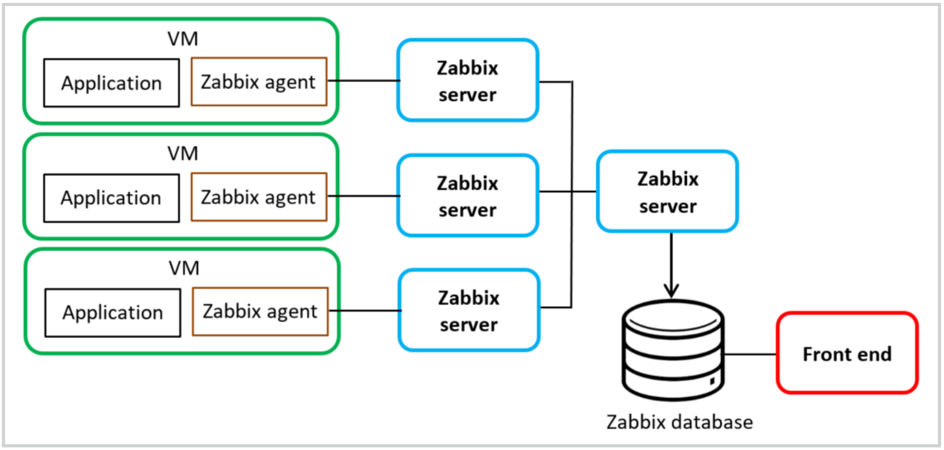


Figure 2.8

Hierarchical architecture of the Zabbix monitoring system.

Lattice [77] is an open-source, non-intrusive monitoring platform which is designed for basically monitoring infrastructure-level parameters in highly dynamic cloud environments, including a large number of resources. The main functionality of the Lattice monitoring system is to collect and distribute the monitoring data through either UDP protocol or multicast addresses. Therefore, this monitoring system does not directly provide the monitoring data to cloud customers, and also it is not aimed at automated alerting and visualisation [78]. The architecture of Lattice is implemented around the concept of producers-consumers. Producers collect monitoring data from monitoring probes which can be whether infrastructure-level probes or application-level probes. Consumers read the monitoring data. The producers and consumers are connected through a network to distribute the measurements collected. The distribution mechanism allows for multiple submitters (called producers) and multiple receivers (called consumers) of monitoring data without having a huge number of network connections. The distribution mechanism can be either IP multicast, Event Service Bus (ESB), or publish/subscribe technique. In each of these distribution mechanisms, a submitter of data needs only to send one copy of a measured value onto the network, and each of the receivers is capable of concurrently collecting the same packet of monitoring data from the network. The basic architecture of the Lattice monitoring system is shown in Fig. 2.9.

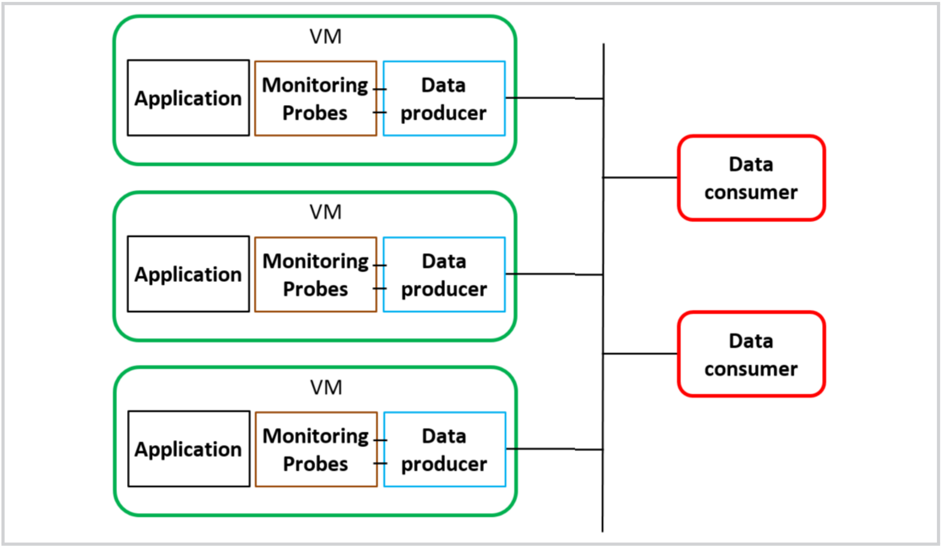


Figure 2.9

Basic architecture of the Lattice monitoring system.

JCatascopia [79] is a scalable monitoring system capable of monitoring federated multi-cloud environments. This open-source monitoring platform is written in Java and published under the Apache 2 license. JCatascopia is designed for monitoring agent/monitoring server architecture. Monitoring agents are capable of measuring whether infrastructure-level metrics (e.g. CPU and memory) or application-specific parameters (e.g. response time and application throughput). Each monitoring agent located on a VM sends the monitoring data to a central Monitoring Server. To this end, each message transmitted to the monitoring server includes the IP address of the monitored resource. Therefore when a change occurs such as live-migration of VMs, the monitoring server is notified, hence and monitoring operations will be continued. Automated alerting capability of JCatascopia has been developed, however this monitoring component is proprietary and not released publically. The basic architecture of the JCatascopia monitoring system is shown in Fig. 2.10.

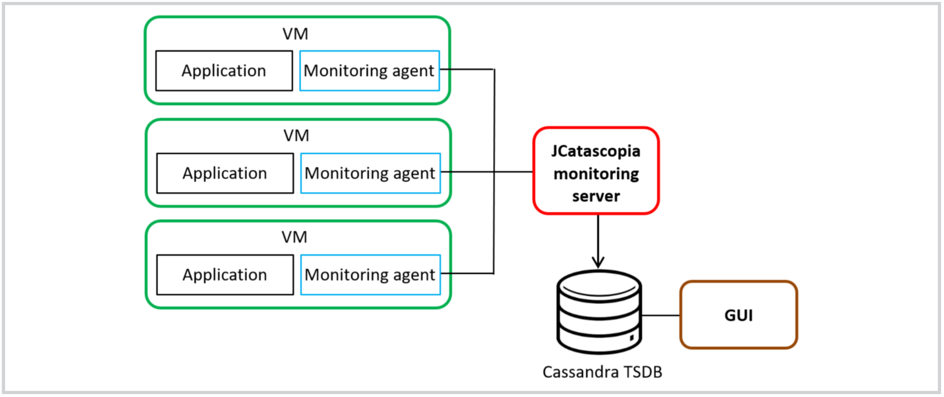


Figure 2.10

Basic architecture of the JCatascopia monitoring system.

2.3 *Developed multi-level monitoring system*

In this thesis, our aim was to develop a multi-level monitoring system capable of monitoring different metrics at both container and application levels. Therefore, in order to implement a monitoring platform used in this thesis for auto-scaling containerised time-critical applications, the JCatascopia monitoring system was selected as the baseline technology and then extended to be able to measure not only application-level metrics, but also container-level metrics.

It should be noted that as a Monitoring Agent in JCatascopia is natively written in Java, each container which includes a Monitoring Agent needs some specific Java packages and a remarkable amount of available memory for a Java Virtual Machine (JVM), although the monitored application running alongside the Monitoring Agent in the container may not be implemented in Java language. Therefore, in order to make the monitoring system based upon a non-intrusive design in this thesis, Monitoring Agents have been developed via the StatsD protocol [80] which can be implemented for many programming languages such as Python and C/C++.

As a consequence, each container in this thesis consists of two parts: (i) an application which represents a service instance and (ii) a Monitoring Agent which represents a StatsD client. The application provides a cloud-based service which processes tasks received from the end-users. The Monitoring Agent is the component which monitors the status of key metrics whether infrastructure-level parameters such as CPU, memory, bandwidth and storage use of the container or application-level metrics such as response time and throughput. Fig. 2.11 shows two different parts of containers co-located on one host.

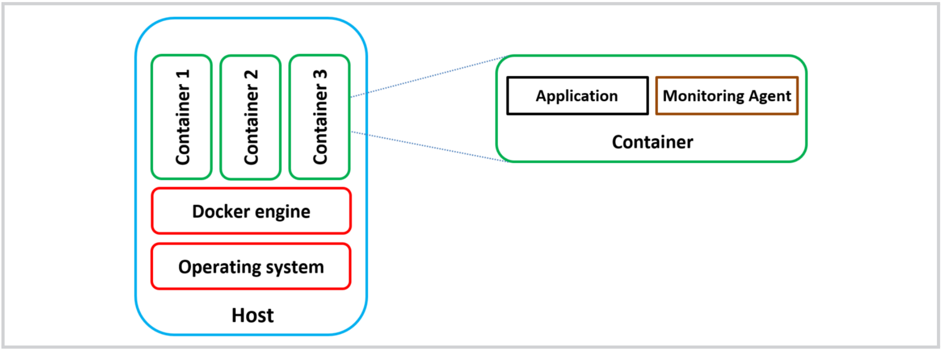


Figure 2.11

Two different parts of containers co-located on one host.

The functioning of the developed monitoring system is illustrated in Fig. 2.12 in which there are two service clusters, namely Cluster 1 and Cluster 2. Each cluster includes containers which provide the same service.

In Fig. 2.12, two container images (illustrated by and) are pulled from a local registry to different hosts. Each container images includes a specific application to be scalable, for instance Service 1 and Service 2. Thus, two different service clusters are shown in this figure. Instantiating a new container instance of a given service means that the service scales up, and terminating a container instance means that the service scales down. Once a new container is started, it means that this container instance is allocated to a logical cluster. The developed monitoring system should keep track of these clusters for every provided service. As an example, Fig. 2.12 shows that Cluster 1 consists of three instances of Service 1, and Cluster 2 hosts two instances of Service 2.

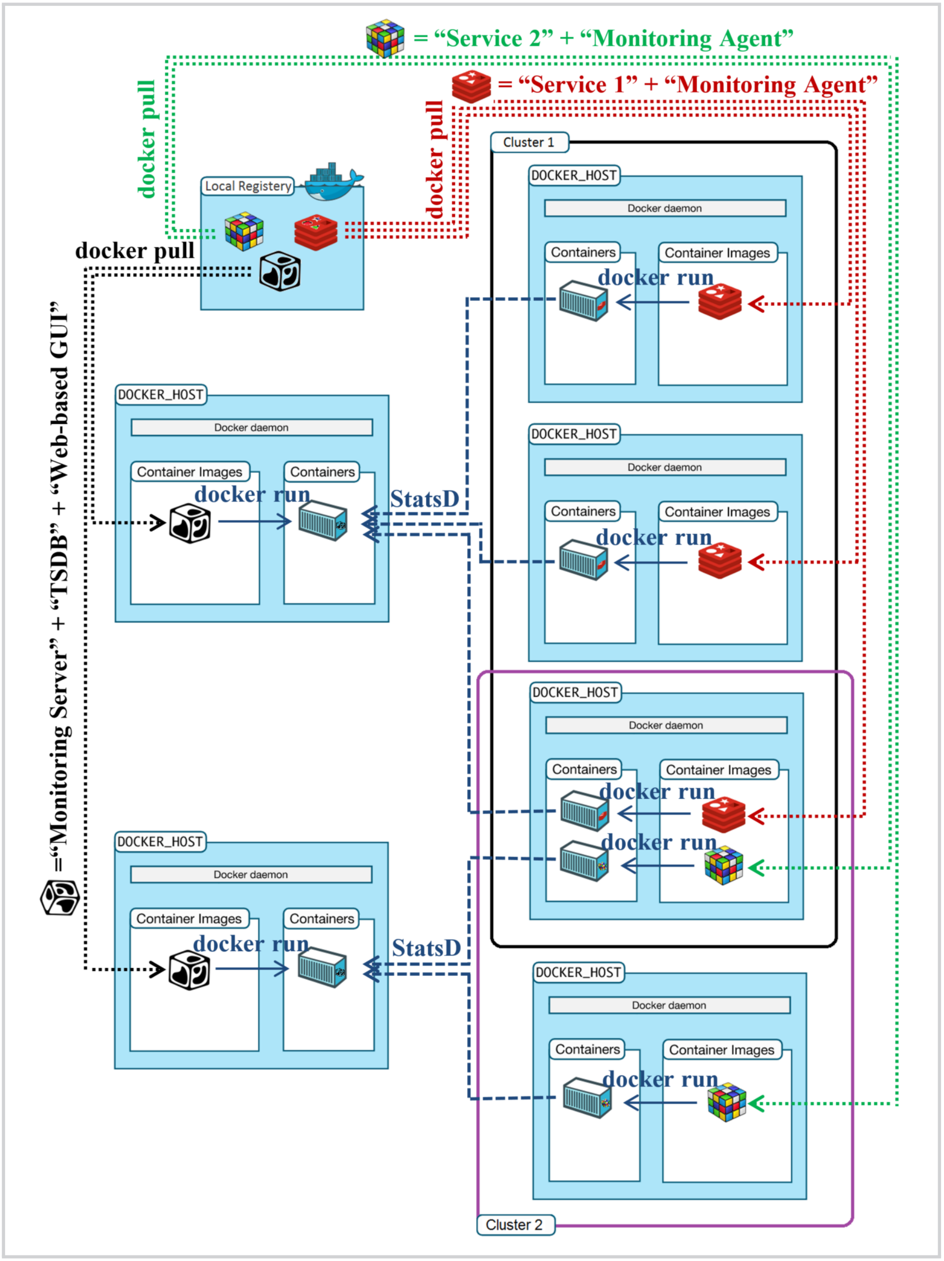


Figure 2.12

Developed monitoring system.

In the developed monitoring system, a container image (illustrated by) is built that contains three different entities: (i) the Monitoring Server which is a StatsD server, (ii) the Cassandra TSDB, and (iii) the Web-based GUI. This container image as an open-source component is publically released on Docker Hub [81]. It must be said that having particular container images built for each of these three entities is also possible. The developed monitoring system is freely accessible to both researchers and commercial companies on GitHub [41] under an Apache 2 license.

In this thesis, a local Docker registry is employed to store and manage all Docker images. In this way by use of a local Docker registry, pulling container images and running container instances of services are much faster across cluster nodes. A local Docker registry remarkably diminishes both network overhead and deployment latency while containers are running across the spread of hosts in a region. Furthermore, it is feasible to specify deployment strategies which take advantage of cached container images, hence further improvement in deployment time.

In order to propose a fine-grained auto-scaling method, our multi-level monitoring system developed to collect information from different levels is more supportive of dynamic adaptation of containerised applications than single-level monitoring techniques. To this end, in addition to monitoring virtualised resources (e.g. CPU, memory, disk, etc.) used by container instances, considering application-level monitoring data is also significant. In Chapter 3, all auto-scaling methods which are currently considered as advanced approaches are explained in detail with a clear rationale to employ monitoring capabilities.

3

State-Of-The-Art Review

This chapter aims at identifying open challenges and understanding limitations of existing methods in auto-scaling of cloud-based applications through a comprehensive literature review. The literature review presented in this chapter covers auto-scaling mechanisms provided by both experience studies and commercial solutions, as summarised in Tab. 3.1. We respectively survey all methods listed in Tab. 3.1 and compare them to identify their characteristics. The differences and similarities among the reviewed methods come up with an opportunity for a thorough conception of the term "auto-scaling" within cloud-based applications. Our proposed method which is called Dynamic Multi-level (DM) auto-scaling is also presented for completeness in the last row of Tab. 3.1.

3.1 Experience research studies

Lorido-Botran *et al.* [82] grouped auto-scaling techniques into five different categories as follows:

- *Control Theory*: Auto-scaling methods which exploit control theory [83–85] create an application performance model used by a component called controller. The controller is able to adjust the necessary amount of resources (e.g. number of application instances) to the application performance at runtime. The controller should be continually improved using a feedback loop. In other words, these methods continuously detect situations when the performance model requires to be upgraded due to the changes in the whole environment and try to accurately correct and adjust the derived performance model.

- *Reinforcement Learning*: Similar to the control theory, reinforcement learning methods [86–88] tend to automate auto-scaling actions, but without leveraging any a *priori* knowledge on the running cloud application such as predefined performance model of the application. On the contrary, reinforcement learning methods aim at learning the most suitable adaptation action with a trial-and-error approach for each particular state on-the-fly. Therefore, such approaches require a considerable time period in order to converge to optimal auto-scaling policies.

- *Queuing Theory*: Queuing Theory can be considered as a mathematical representation of waiting lines. Auto-scaling methods which apply queuing theory [89, 90] can adjust the resource capacity by making decisions based upon the size of requests inside the queue. For example, service requests sent from the end-users are queued at the load-balancer. Given the workload rate at runtime, such methods try to achieve the optimal amount of required resources in order to serve maximum requests in the queue with an acceptable response time.

- *Time-series Analysis*: Auto-scaling approaches which use time-series analysis methods try to detect patterns and forecast future values based on sequences of data points. In order to achieve a great level of accuracy, a right time-series analysis method has to be applied. Moreover, the prediction interval and history window should be correctly chosen. For example, the DTW algorithm [91] can be used to recognise patterns in time series of data points. A DTW-based similarity checking mechanism measures the difference between two time-series, namely the incoming monitoring data and the reference patterns. DTW is an algorithm to provide a similarity measure between two signals that also allows for stretched and compressed sections of two given sequences. In other words, the main advantage of DTW is its ability to automatically cope with time deformations of two signals corresponding to execution environments performed at two different velocities. In this way, reference patterns as templates should be predetermined for each adaptation event such as scaling up or scaling down. Therefore, sufficient data set needs be collected in advance before the system starts working.

- *Threshold-based Rules*: Threshold-based auto-scaling rules are generally offered by commercial cloud-based infrastructure providers. These rules are considered as purely static, reactive auto-scaling policies. In this way, adaptation actions are commenced according to the value of some performance parameters based upon a set of predefined thresholds. The most important advantage of such threshold-based auto-scaling approaches are their simplicity since these rules are easy-to-set-up for the scalability of underlying cloud-based infrastructures. Our new threshold-based auto-scaling method proposed in this thesis is dynamic because this method uses dynamic thresholds instead of fixed auto-scaling rules provided by current commercial cloud-based infrastructure providers.

Al-Sharif *et al.* [92] proposed a framework named Autonomic Cloud Computing Resource Scaling (ACCRS) in order to provision a sufficient number of VMs to address the changing resource requirements of an application running on the cloud. The proposed adaptation method employs a set of fixed thresholds for infrastructure-level metrics. These thresholds are used for CPU, memory, and bandwidth utilisation to evaluate runtime states of resources. In this approach, the workload can be identified as a light or heavy weight if any of these attributes violate the associated thresholds. The presented resource scaling framework uses a single-level monitoring system able to measure only infrastructure-level metrics. Therefore, the service response time and application throughput do not have any role in specifying the necessary auto-scaling actions.

Islam *et al.* [93] presented a proactive cloud resource management system in which neural networks and linear regression have been used to predict and satisfy future resource demands. The developed performance prediction model is able to estimate upcoming resource usage (such as an aggregated percentage of CPU utilisation of all running VM instances) at runtime and then launches additional VMs to maximise application performance. In this work, only CPU usage is employed to train the prediction model, and their method does not involve other types of resources, for example memory. This approach uses a 12-minute prediction interval, because the setup time of VM instances is almost 5 to 15 minutes in general. This low rate of prediction is not satisfactory for continuously changing workloads. Furthermore, in such proactive methods [94–97], for each new change in the workload, it takes too long to converge towards a stable driven performance model, and therefore the application may deliver poor service quality to the end-users during the beginning stages of the learning period.

Jamshidi *et al.* [86] proposed a self-learning adaptation technique called FQL4KE to perform auto-scaling operations in order to increase or decrease the number of application instances built upon VM-based virtualisation. FQL4KE uses a fuzzy control approach based on a reinforcement learning method. Although, there are real-world environments in which the number of situations is enormous, and therefore the reinforcement learning method may take too long to converge for any new change in the execution environment. Hence, applying reinforcement learning method may become impractical due to the time constraints imposed by some time-critical applications such as disaster early warning systems.

Table 3.1

Overview of existing auto-scaling methods used for cloud-based applications

Method	Virtualisation technology	Infrastructure-level metrics	Application-level metrics	Technique	Adjustment ability
Al-Sharif *et al.* [92]	VM	CPU, memory, and bandwidth	Nothing	Rule-based	Static
Islam *et al.* [93]	VM	CPU	Response time	Linear regression and neural networks	Static
Jamshidi *et al.* [86]	VM	CPU, memory, etc.	Response time and application throughput	Reinforcement learning (Q-Learning)	Dynamic
Arabnejad *et al.* [87]	VM	Nothing	Response time and application throughput	Fuzzy logic control and reinforcement learning	Dynamic
Tsoumakos *et al.* [88]	VM	CPU, memory, bandwidth, etc.	Response time and application throughput	Reinforcement learning (Q-Learning)	Static
Gandhi *et al.* [89]	VM	CPU	Response time and application throughput	Queueing model and Kalman filtering	Dynamic
Kukade and Kale [98]	Container	Memory	Application throughput	Rule-based	Static
Kan [99]	Container	CPU and memory	Application throughput	Rule-based and ARMA	Static
Baresi *et al.* [83]	Container	CPU and memory	Response time and application throughput	Control theory	Dynamic
Qu *et al.* [100]	VM	CPU, memory, bandwidth, etc.	Nothing	Profiling	Static
Horizontal Pod Auto-scaling (HPA) used by Kubernetes	Container	CPU	Nothing	Rule-based	Static
Target Tracking Scaling (TTS) and Step Scaling (SS) used by Amazon	VM and container	CPU and bandwidth	Application throughput	Rule-based	Static
THRESHOLD (THRES) [101]	VM and container	CPU	Nothing	Rule-based	Static
Multiple Policies (MP) used by Google.	VM	CPU	Application throughput	Rule-based	Static
Dynamic Multi-level (DM)	Container	CPU, memory and bandwidth	Response time and application throughput	Rule-based	Dynamic

Arabnejad *et al.* [87] developed a fuzzy auto-scaling controller which is combined with two types of reinforcement learning approaches: (1) Fuzzy Q-learning Learning (FQL) and (2) Fuzzy SARSA Learning (FSL). In this work, the monitoring system collects different metrics required such as service response time, application throughput, and the number of VMs in order to feed the proposed auto-scaling controller. The auto-scaling controller is able to automatically scale the number of VMs for dynamic resource allocations to react to workload variations. It should be noted that the presented architecture is operational only for a certain kind of virtualisation platform called OpenStack. Besides, the controller needs to choose scaling operations among a limited number of possible actions. In other words, if a drastic surge suddenly appears in the workload intensity, the presented auto-scaling method can add only one or two VM instances that probably cannot offer enough resources to maintain a favourable QoS.

Tsoumakos *et al.* [88] presented a resource provisioning approach which is called TIRAMOLA in order to identify the number of VM instances necessary to meet user-defined objectives for a NoSQL database cluster. The proposed technique combines Markov Decision Process (MDP) with Q-learning as a reinforcement learning method. This resource provisioning approach continuously decides the most beneficial state that can be achieved at runtime, and thus specifies possible actions in each state that can either do nothing, or add/remove NoSQL nodes. The principle of TIRAMOLA is acting in an expected style of operation when the regular workload scenario can be specified. Accordingly, previously unobserved workloads are considered as the major obstacle to the fast adaptation of the entire system to meet the performance objective of cloud-based interactive services. Furthermore, TIRAMOLA is restricted to the elasticity of a specific type of application such as NoSQL databases. Moreover, the monitoring part needs to collect client-side statistics in addition to server-side metrics (e.g. CPU, memory and bandwidth, query throughput, and so on). In this regard, clients of such applications should be modified so that each one can report its own monitoring statistics, which is not an operational solution for many real-world use cases.

Gandhi *et al.* [89] proposed a model-driven auto-scaler which is called Dependable Compute Cloud (DC2). This auto-scaler proactively has a tendency to ensure the QoS of the application in order to meet user-specified requirements. The presented method uses a combination of the Kalman filter technique and a queueing model to provide estimations of the average service time at runtime. The functionality of DC2 is emphasised on avoiding under-utilisation of resources, and thus it may result in an over-provisioning problem during execution time. Moreover, the Kalman filter process is repetitively continued at every 10-second monitoring interval, it requires some time (e.g. few minutes) to calibrate the driven model based on the monitoring information for each new state. Therefore, the challenge in this regard lies in the accuracy of the presented auto-scaling approach which may shrink for special workload scenarios such as a new, drastically varying pattern over time.

Kukade and Kale [98] demonstrated a master-slave auto-scaling architecture for containerised applications. Slaves represent the nodes where container instances can be deployed, while there is a master that is responsible for receiving arrival requests and routing them to running containers. The master also includes a self-adapter module that is able to check two different single-level scaling rules in order to increase or decrease the number of running containers. For example, if the request rate exceeds a pre-defined fixed threshold, a new container instance will be started. If the memory load of containers reaches a threshold, then a new replica of container instance will be launched. However, in a real-world auto-scaling platform, monitoring additional metrics usable within adaptation rules is necessary. For example, CPU utilisation which is important for computing-intensive applications has not been considered by the study. Kan [99] introduced a container-based elastic cloud platform called DoCloud. This platform incorporates proactive and reactive models to calculate the number of containers to be added for the scale-out, while only the proactive model is applied for the scale-in. DoCloud exploits static thresholds for CPU and memory utilisation, and uses the Autoregressive-Moving Average (ARMA) method to predict the number of incoming requests for the application.

Baresi *et al.* [83] proposed an auto-scaling approach that employs an adaptive discrete-time feedback controller which enables a container-based application to dynamically scale necessary resources, both horizontally and vertically. Horizontal scaling means the addition or removal of container instances, while vertical scaling represents increasing or decreasing the amount of resources allocated to a running container. In this work, a component called ECoWare agent needs to be deployed in each VM. An ECoWare agent is in charge of the collection of container-specific monitoring data, such as containers' usage of CPU, memory, etc. This component is also responsible for instantiating or terminating a container in the VM, or changing the resources assigned to a container.

Qu *et al.* [100] proposed an auto-scaling method which exploits heterogeneous VM instances in order to provision web applications. The intention of using heterogeneous

VM instances in this work is improving the reliability of service clusters as well as resource cost. First of all, the proposed auto-scaling approach observes multiple infrastructure-level metrics such as CPU, memory, etc. Then, the auto-scaler will profile the application regarding its average resource consumption for different workload conditions. The profiling step in this work should be performed offline, however the authors claim that the proposed approach is open to be extended for dynamic online profiling. With the profile driven from the previous step, the auto-scaling method is capable of estimating the processing capability of each spot VM instance under varied amount of workload. Therefore, it is possible to decide how to distribute incoming requests to heterogeneous VMs in order to balance their loads. However, this work is restricted to mainly fault-tolerant applications.

3.2 *Existing auto-scaling production solutions*

The current commercial cloud-based infrastructure providers (e.g. Google Cloud Platform and Amazon EC2) as well as open-source container management platforms (e.g. Kubernetes) offer static rule-based auto-scaling methods which are not flexible in order to adjust themselves to dynamic changes of the operational environment. In this section entitled "Existing auto-scaling production solutions", some widely used rule-based auto-scaling approaches are explained for the purpose of comparison to the proposed Dynamic Multi-level (DM) auto-scaling method. These approaches are chosen for comparison to our DM method because they are also rule-based and considered as advanced auto-scaling solutions, which are used in the current production systems. Fig. 3.1 presents two significant quality features which are inspected by this thesis and result in the definition of a fine-grained auto-scaling method.

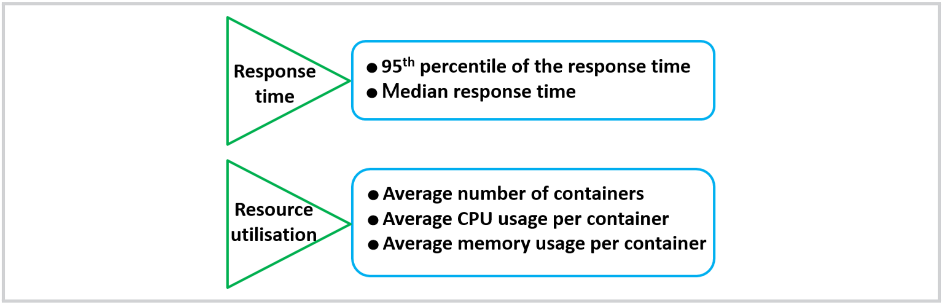

Response time
● 95th percentile of the response time
● Median response time
Resource utilisation
● Average number of containers
● Average CPU usage per container
● Average memory usage per container

Figure 3.1

Significant quality properties of cloud-based applications.

An ordinary practice in the existing commercial cloud-based infrastructure is to employ fixed, infrastructure-level auto-scaling rules. For instance, a CPU-based auto-scaling policy can be specified in a way that more VMs/containers should be instantiated if the average CPU utilisation reaches a fixed threshold such as 80%; while some VMs / containers may be stopped if the average CPU utilisation is below 80%. These auto-scaling configurations cannot be very helpful for certain workload scenarios for example drastically changing patterns. Furthermore, these settings lead to a stable system at the best case towards 80% resource utilisation that means 20% of resources are wasted, which is not desired. An important challenge in providing an auto-scaling approach is to what extent this mechanism can be self-adjustable to changing conditions in the operational environment.

In the proposed new DM auto-scaling method, both infrastructure-level metrics (e.g. CPU and memory) as well as application-level metrics (e.g. response time and application throughput) are the factors which can dynamically influence the adjustable auto-scaling rules. Our proposed method is dynamic because this method applies self-adaptive auto-scaling rules which are used for instantiating and terminating container instances. These auto-scaling rules are adjusted at runtime according to the workload intensity. In other words, in our auto-scaling method, conditions when containers are launched or terminated can be different and do not have to be predefined. In the following subsections, an analysis of existing widely used auto-scaling methods which serve as means for comparison to our proposed DM method.

3.2.1 Kubernetes - Horizontal Pod Auto-scaling (HPA)

Kubernetes is a lightweight open-source container management platform which is capable of orchestrating containers and automatically providing horizontal scalability of running applications. In Kubernetes, a group of one, or a small number of containers which are tightly coupled together with a shared IP address and port space can be defined as a pod. Therefore, a pod simply indicates one single instance of an application which can be replicated, if more instances are helpful to handle the increasing workload. In Kubernetes, an auto-scaling approach which is called Horizontal Pod Auto-scaling (HPA) [102] is a control loop algorithm basically based upon only CPU utilisation; no matter how workload is changing or application performance is behaving. HPA which shown in Fig. 3.2 can increase or decrease the number of pods to keep the average CPU utilisation across all pods at, or close to, a target value such as 80%.

```
Kubernetes HPA algorithm
Inputs:
$Target_{cpu}$: Targeted per-pod CPU resource usage
CLTP: Control Loop Time Period in seconds, e.g. 30 seconds
Outputs:
NoP: Number of Pods to be running
do{
    Cluster = [Pod1,..., PodN];
    SumCpu=SUM_Cluster(cpu_usage_of_pod1,...,cpu_usage_of_podN);
    NoP = $\left\lceil \frac{SumCPU}{Target_{cpu}} \right\rceil$
    wait(CLTP);
} while(true);
```

Figure 3.2

Kubernetes Horizontal Pod Auto-scaling (HPA) algorithm.

In HPA, SUM_Cluster is the grouping function employed for calculating the total sum of the CPU utilisation of the cluster. The auto-scaling period of the Kubernetes auto-scaler is half a minute (30 seconds) by default that can be changed. At each auto-scaling iteration, Kubernetes' controller may add or remove the number of pods according to Number of Pods (NoP) to be running that is the output of the HPA algorithm.

3.2.2 Amazon EC2 AWS - Target Tracking Scaling (TTS)

The Amazon EC2 AWS platform provides an auto-scaling method called Target Tracking Scaling (TTS) [103] mechanism, which is capable of offering dynamic adjustments in the number of instances based on a target value for a specific metric. The TTS mechanism uses single-level auto-scaling rules considering either an infrastructure-level metric (e.g. average CPU utilisation) or an application-level metric (e.g. application throughput per instance). In this regard, a predetermined target value should be set for a metric used in the auto-scaling rule. Moreover, the minimum and maximum number of instances in the service cluster must be defined. The TTS method increases or decreases application instances when it is necessary in order to keep the metric at, or close to, the predefined target value.

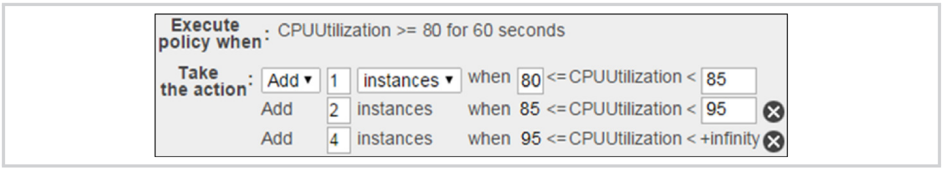
The default settings in AWS are able to scale the application based on the value of a metric with a 5-minute frequency. This frequency can be changed to 60 seconds (one minute) that is known as detailed auto-scaling option in the Amazon EC2 AWS platform. TTS is able to increase the cluster size when the specified metric is above the target value, or decrease the cluster capacity when the specified metric is less than the target value for specified consecutive periods e.g. even one interval. For a large cluster, the workload is spread over a large number of instances. Therefore in such situation, adding a new instance to the cluster or removing a running instance from the cluster causes less of a gap between the target value and the actual metric data points. On the contrary, for a small cluster in which the number of instances is not big, adding or removing an instance may result in a big gap between the predefined target value and the actual metric data points. As a consequence, together with keeping the value of the metric close to the target value, the TTS method must also adjust itself to reduce rapid changes in the cluster size.

As an example, an auto-scaling rule predefined as "TTS1 (CPU, 80%, ±1)" can be taken into account to keep the average CPU utilisation of the whole cluster at 80% by instantiating or terminating one instance per scaling action. Furthermore, the Target Tracking Scaling rule 'TTS' may also be specified to adjust the number of instances by a percentage. In this way for example, an auto-scaling rule named "TTS2 (CPU, 80%, ±20%)" is able to add 20% more instances or terminate 20% fewer instances, if the conditions are met. In order to give an example, if four instances are currently running in the cluster, and the average CPU utilisation goes over 80% during the last minute, TTS2 determines that 0.8 instance (that is 20% of four instances) should be added to the cluster. In such a case, TTS rounds up 0.8 and hence instantiates one new instance. Or, if in a certain condition, TTS2 decides to remove 1.5 instances, TTS can round down and remove only one instance from the cluster.

3.2.3 Amazon EC2 AWS - Step Scaling (SS)

Amazon EC2 AWS also provides another type of auto-scaling approach which is called Step Scaling (SS) [104]. If the average CPU utilisation requires being below a threshold, for example 80%, it can be possible to specify different scaling steps used for auto-scaling application. Fig. 3.3 presents the first part of an AWS Step Scaling example called 'SS1' to increase the capacity of the cluster, when the workload is growing. In this example, one instance will be instantiated for a modest breach (from 80% to 85%), two more instances will be added for somewhat bigger breaches (from 85% to 95%), and lastly four instances will be instantiated for CPU utilisation that exceeds 95%. It should be noted that the ranges of step adjustments should not overlap with each other or even have a gap between them. In this example of Step Scaling, SS1 periodically measures the one-minute aggregated value of the average CPU utilisation from all instances. Subsequently, if this value exceeds the threshold 80%, SS1 compares it against the lower and upper bounds specified by different step adjustments to determine which auto-scaling action needs to be executed.

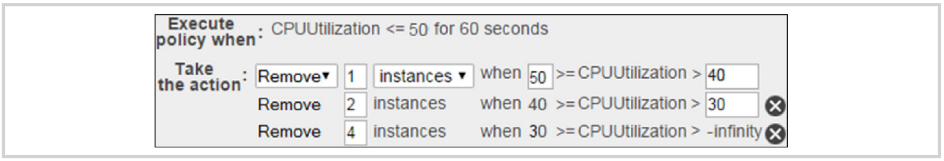
In similar way, defining various steps to reduce the number of instances running in the cluster is also possible, when the workload is decreasing. As an example, Fig. 3.4 shows three different steps to stop unnecessary instances when the average CPU utilisation falls below the threshold 50%.



```
Execute
policy when: CPUUtilization >= 80 for 60 seconds

Take
the action: Add ▾  1  instances ▾  when 80 <= CPUUtilization < 85
            Add    2  instances    when 85 <= CPUUtilization < 95
            Add    4  instances    when 95 <= CPUUtilization < +infinity
```

Figure 3.3 AWS auto-scaling example named SS1 in order to increase the cluster size.



```
Execute
policy when: CPUUtilization <= 50 for 60 seconds

Take
the action: Remove▾  1  instances ▾  when 50 >= CPUUtilization > 40
            Remove   2  instances    when 40 >= CPUUtilization > 30
            Remove   4  instances    when 30 >= CPUUtilization > -infinity
```

Figure 3.4 AWS auto-scaling example named SS1 in order to decrease the cluster size.

In Amazon EC2 AWS, Step Scaling policies can be also specified on a percentage basis. That means in order to handle an increasing workload at runtime, Step Scaling approach is capable of increasing the number of instances by the percentage of the current cluster size. Fig. 3.5 shows the first part of an AWS Step Scaling example called 'SS2' that consists of two step adjustments to increase the number of instances running in the cluster by 20% and 30% of the cluster size at the respective steps. If the resulting value is not an integer, SS2 will round this value. In this case, values greater than 1 will be rounded down. Resulting values between 0 and 1 will be rounded to 1. For example, if the current number of instances in the cluster is four, adding 30% of the cluster will result in the deployment of one more instance. Furthermore, 20% of four instances is 1.2 instances, which is rounded down to one instance.

Determining an identical set of Step Scaling policies to decrease the number of instances deployed in the cluster is also possible. In such a way, SS2 method is able to decrease the current capacity of the cluster by the defined percentage at different step adjustments. Fig. 3.6 shows a two-step auto-scaling approach to handle a decreasing workload at runtime, and hence to dynamically reduce the number of instances in the cluster by 20% and 30% of the cluster size. The resulting values between 0 and -1 will be rounded to -1. Additionally, the values less than -1 will be rounded up. For example, -3.78 is rounded to -3.

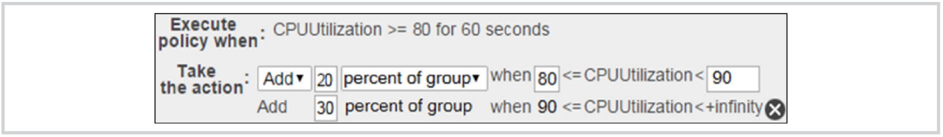

Execute policy when: CPUUtilization >= 80 for 60 seconds
Take the action: Add 20 percent of group when 80 <= CPUUtilization < 90
Add 30 percent of group when 90 <= CPUUtilization < +infinity

Figure 3.5
AWS auto-scaling example named SS2 in order to increase the cluster size.

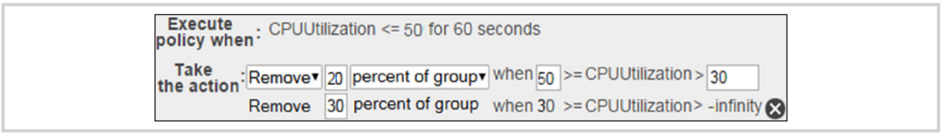

Execute policy when: CPUUtilization <= 50 for 60 seconds
Take the action: Remove 20 percent of group when 50 >= CPUUtilization > 30
Remove 30 percent of group when 30 >= CPUUtilization > -infinity

Figure 3.6
AWS auto-scaling example named SS2 in order to decrease the cluster size.

3.2.4 First method of THRESHOLD (THRES1) and second method of THRESHOLD (THRES2)

A static single-level auto-scaling method which is called THRESHOLD or THRES (Metric, UP%, DOWN%) [101] can horizontally add a container instance if an aggregated metric (such as the average CPU or memory utilisation of the cluster) reaches the predetermined UP% threshold. Moreover, it is able to remove a container instance when it falls below the predefined DOWN% threshold for a default number of consecutive intervals, e.g. two intervals. The approach named "THRES1 (CPU, 80%, 50%)" is an example for such a static single-level auto-scaling method.

The "THRES2 (CPU, 80%, 50%, RT, 190 ms)" method also can be determined as an example for a static multi-level provisioning approach which is capable of considering the average Response Time (RT), in addition to the average CPU utilisation of the cluster. In order to add a new instance, both the average CPU resource utilisation and response time thresholds (in this example, 80% and 190 ms, respectively) require to be violated for two successive intervals. In order to remove a running instance from the cluster, the average CPU utilisation of the cluster needs to be below 50% during the last two consecutive periods.

3.2.5 Google Cloud Platform - Multiple Policies (MP)

The Google Cloud Platform offers an auto-scaling mechanism which is called 'MP' using multiple auto-scaling policies individually at different levels of monitoring, such as infrastructure and application-level metrics [105]. For example, the MP auto-scaler is capable of considering two policies. One policy can be on the basis of average CPU usage of the cluster that is an infrastructure-level metric. Another policy can be on the basis of Application Throughput of the Load-Balancer (ATLB) that is an application-level metric. It means that, each of these two policies is a single-level rule that is defined and based upon only one parameter. MP calculates the number of required instances recommended by each policy, and afterwards selects the policy that defines the largest number of instances in the cluster. This characteristic of the MP auto-scaling approach conservatively ensures that the cluster always has enough capacity to handle the workload over time.

In this manner, a target value needs to be specified for each metric. For instance, "MP (CPU=80%, ATLB=80%)" is a two-policy auto-scaling method which continuously monitors both the average CPU usage of the cluster, and also the load-balancing serving capacity. In this example, setting a 0.8 target utilisation designates the MP auto-scaler to keep the average CPU utilisation at 80% in the cluster. Furthermore, MP scales the cluster to maintain 80 percent of the load-balancing serving capacity since ATLB is 80%. For example, if the maximum load-balancing serving capacity is defined as 100 Requests Per Second (RPS) per instance, MP will add or remove instances from the cluster to maintain 80% of the serving capacity, or in other words 80 RPS per instance.

Detailed explanation of auto-scaling approaches presented in this chapter offers a meaningful interpretation and insight into our proposed new method called Dynamic Multi-Level (DM) auto-scaling described in Chapter 4.

Method and Architecture

This thesis presents a new Dynamic Multi-level (DM) auto-scaling method which is implemented to be used according to our proposed new architecture for auto-scaling containerised time-critical applications. The DM auto-scaling method exploits our developed multi-level monitoring system explained in Chapter 2. The use of the DM auto-scaling method in an innovative functional architecture is shown in Fig. 4.1 for adaptive containerised applications.

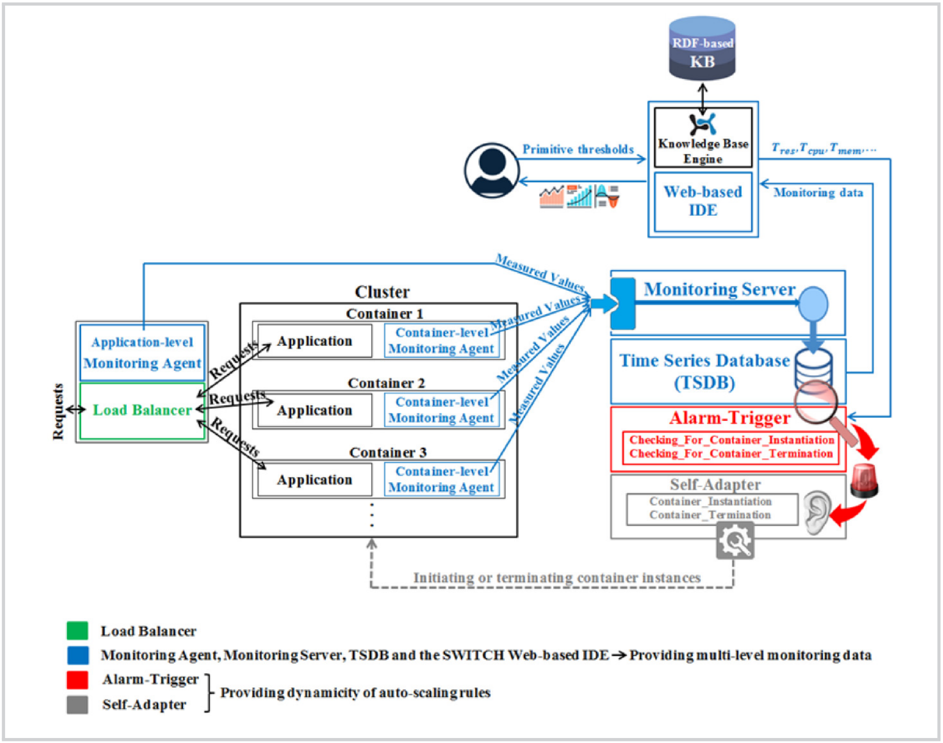

RDF-based KB
Knowledge Base Engine
Primitive thresholds
Web-based IDE
$T_{res}, T_{cpu}, T_{mem}, \ldots$
Monitoring data
Measured Values
Monitoring Server
Cluster
Container 1
Application
Container-level Monitoring Agent
Application-level Monitoring Agent
Requests
Load Balancer
Container 2
Container 3
Time Series Database (TSDB)
Alarm-Trigger
Checking_For_Container_Instantiation
Checking_For_Container_Termination
Self-Adapter
Container_Instantiation
Container_Termination
Initiating or terminating container instances
🟩 Load Balancer
🟦 Monitoring Agent, Monitoring Server, TSDB and the SWITCH Web-based IDE → Providing multi-level monitoring data
🟥 Alarm-Trigger
⬛ Self-Adapter
Providing dynamicity of auto-scaling rules

Figure 4.1

Proposed architecture for adaptive containerised applications.

In this thesis, it is considered that each host in a cluster can comprise at most one container instance per service, while each host can belong to various clusters at the same time. In other words, more than one container can be located on one host machine, but nevertheless they have to provide different services. This condition is a realistic situation for an operational environment where various types of services need to be scaled. When a particular service is launched at the host, it exposes its associated interfaces at certain port numbers, which should not clash with the port numbers of other running services on that host. In this way, it is meaningful to provide an internal, so-called vertical elasticity technique for the allocation of CPU and memory resources to different services within the same host, but, it makes no sense to launch additional container instances of the same service on the same host machine.

In general term, if two or more container instances are co-located on one host, by default all container instances will achieve the same proportion of CPU cycles. In this case, if tasks in a container are idle, other containers are capable of using the leftover CPU cycles. Besides this, it is feasible to modify identical proportions allocated to running containers by using a relative weighting mechanism. In this way, when all containers running on one host try to achieve all 100% of the CPU time, the relative weights give each container access to an assigned proportion of the host machine's CPU cycles (since CPU cycles are limited).

In conditions when enough CPU cycles are available, all container instances running on a host machine are able to use as much CPU as they require regardless of the dedicated weights. Although, there is no guarantee that each container can use a specific amount of CPU time at runtime. This is because the actual amount of CPU cycles assigned to each container will vary depending on the number of container instances co-located on the same host machine and also the relative CPU-share settings allocated to containers. In order to make sure that no container can starve out other containers on a single host machine, if a running container instance provides a CPU-intensive service, other container instances that will be deployed on that host machine should not be identified as computationally-intensive services. This concept has been employed also for memory-intensive services. In this thesis, all containers have the same weight to gain access to the CPU cycles and the same limit at the use of memory. This makes it a suitable case of so-called horizontal scaling.

The proposed architecture using the DM auto-scaling method consists of various components, namely: (1) Load-Balancer, (2) Monitoring Agent, (3) Monitoring Server, (4) Time Series Database (TSDB), (5) Web-based Interactive Development Environment (IDE), (6) Knowledge Base (KB) Engine and Resource Description Framework (RDF)-based KB, (7) Alarm-Trigger and (8) Self-Adapter. These components are respectively explained in detail in the next sections.

4.1 Load-Balancer

The Load-Balancer, such as HAProxy [106], provides high-availability support for cloud-based applications by spreading requests across multiple instances. HAProxy, shown in Fig. 4.2, is a free, open-source, fast and reliable software which offers load balancing mechanism. In this thesis, the HAProxy is used to drive the actual load balancing of requests among multiple container instances.

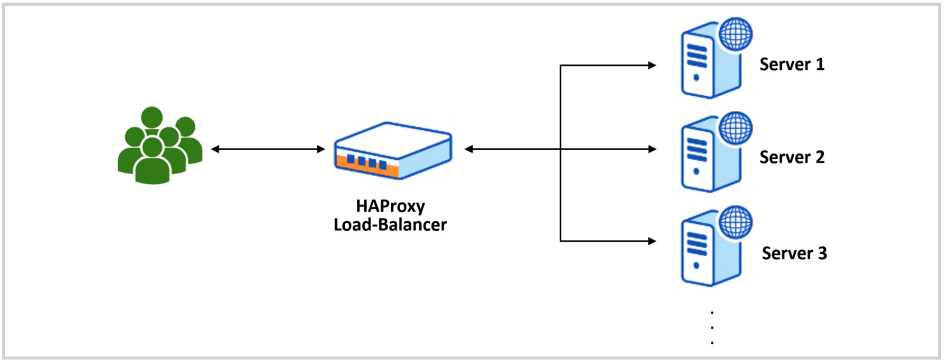


Figure 4.2

HAProxy Load-Balancer.

HAProxy is widely used by a number of auto-scaling research works [107–117] and also high-profile commercial solutions including GoDaddy [118], GitHub [119], Stack Overflow [120], Reddit [121], Speedtest [122], Bitbucket [123], Twitter [124], W3C [125] as well as the AWS OpsWorks [126] product from Amazon Web Services.

4.2 *Monitoring Agent*

The monitoring system is capable of measuring both container-level metrics (e.g. CPU and memory usage of containers) and application-level parameters (e.g. average response time and throughput of the application). Thus, two types of Monitoring Agents which monitor container-level and application-level metrics are considered in the architecture as shown in Fig. 4.3.

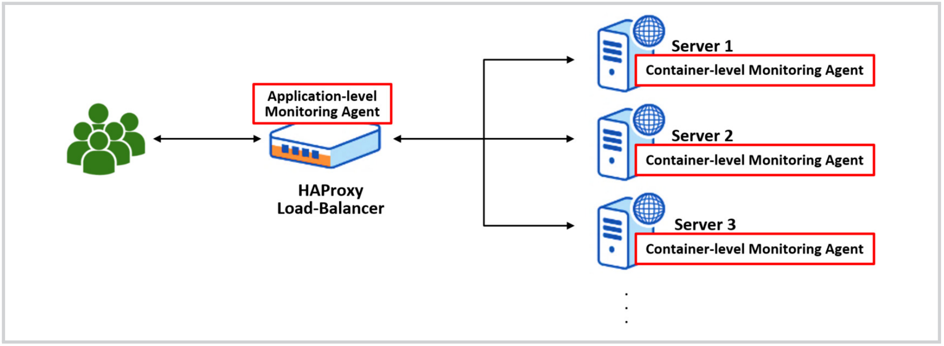

Application-level
Monitoring Agent
HAProxy
Load-Balancer
Server 1
Container-level Monitoring Agent
Server 2
Container-level Monitoring Agent
Server 3
Container-level Monitoring Agent

Figure 4.3

Two types of Monitoring Agents to monitor container-level and application-level metrics.

As mentioned before, every host can be determined to locate more than one container in a way that each container provides different service, or in other words containers co-located on one host belong to different cluster. Therefore as an example, if a host consists of two containers, two container-level Monitoring Agents will be running on the host as shown in Fig. 4.4. This is because every container includes a container-level Monitoring Agent which measures CPU and memory utilisation of the container.

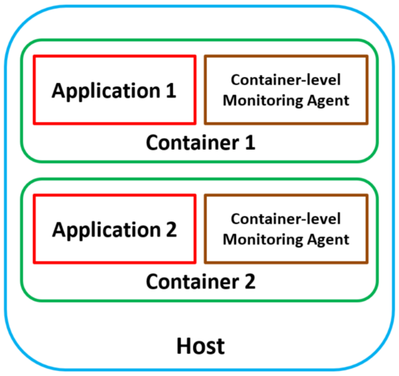

Application 1
Container-level Monitoring Agent
Container 1
Application 2
Container-level Monitoring Agent
Container 2
Host

Figure 4.4

Each container includes a container-level Monitoring Agent.

In this thesis, the application-level Monitoring Agent is responsible for monitoring the Load-Balancer. Application-level metrics which are measured in the context of the proposed DM auto-scaling method are *AvgRT* (average response time to reply to an end-user's request), *AT* (application throughput which means the average number of requests per second processed by one container instance), and *cont* (number of container instances behind the Load-Balancer).

The distributed principle of our implemented agent-based monitoring platform supports an interoperable, lightweight architecture which can quench the runtime overhead of the whole monitoring system only to a number of Monitoring Agents. A Monitoring Agent which is running alongside the application in a container instance collects individual metrics and aggregates the measured values in order to be sent to the Monitoring Server.

4.3 Monitoring Server

The Monitoring Server represents a component which receives measured metrics from the Monitoring Agents, whether container-level or application-level Monitoring Agents.

This monitoring system is capable of storing measured values in the Apache Cassandra server as TSDB. It takes almost one minute to run the Monitoring Server. The Monitoring Server should be running on a machine with enough memory, disk and CPU resources. This machine should address the Cassandra hardware requirements. To this end, the minimal requirements could be at least 2-core CPU with minimum 8/16 GB of RAM and 60 GB of storage.

When a container instance is instantiated, the container-level Monitoring Agent will automatically send the Monitoring Server a message in order to register itself as a new metric stream, and afterwards it will start collecting monitoring metrics (e.g. CPU and memory usage of the container) and continuously forward the measured values to the Monitoring Server.

4.4 Time Series Database (TSDB)

In this thesis, Cassandra which is a free, open-source, column-oriented, NoSQL database system is used to store the time series data. Cassandra database is designed to store and handle large amount of data.

It has its own query language called the Cassandra Query Language (CQL). CQL which is an alternative to the traditional Structured Query Language (SQL) can be considered as a simple interface in order to access Cassandra. CQL adds an abstraction layer to hide implementation details and provides native syntaxes for collections and other common encodings. Language drivers are available for Java (JDBC), Python (DBAPI2), Node.JS (Helenus), Go (GOCQL) and C++.

4.5 Web-based Interactive Development Environment (IDE)

A Web-based IDE is used to set primitive thresholds which are needed for adaptation policies. This component allows for all external entities to access the monitoring information stored in Time Series Database in a unified way, via pre-prepared REST-based APIs. It therefore represents an intermediate layer between information consumers and information provider, and prevents security issues if APIs were accessed via external entities.

Since the Web-based IDE shows runtime monitoring information, it is also a key tool used by software engineers to analyse events in a dynamically changing cloud environment.

4.6 *Knowledge Base (KB) Engine and RDF-based KB*

The Knowledge Base Engine component is in charge of all operations that control the collection of various values (e.g. auto-scaling rules, characteristics of virtualisation platforms, features of hosts, primitive thresholds, their unit, their data type, their range, periodic application QoS, etc.) as RDF triples, along with actually storing and also retrieving these data on disk. The Knowledge Base Engine component continuously stores information about the execution environment in a Knowledge Base (KB) which may be used for interoperability, analysing, integration and optimisation purposes. Maintaining such a KB allows analysis of long-term trends, supports capacity planning and enables for a variety of strategic analysis for example year-over-year comparisons and usage trends. The Knowledge Base is also aimed at addressing information integration requirements of stakeholders, such as cloud-based service providers, allowing for flexible data movement and seamless information management solution.

The Knowledge Base Engine component is implemented by using a Jena Fuseki [127] server to load an RDF dataset and make it accessible via a REST API as a SPARQL endpoint, in order to expose the CRUD operations for Creating, Retrieving, Updating and Deleting RDF records. Jena Fuseki is an open-source, lightweight database server which is easy to be installed, and it is capable of efficiently storing large numbers of RDF triples on disk [128].

A key characteristic of the Knowledge Base design is the notion of interchangeability. In order to ensure interchangeability, it is necessary that the interaction between information objects be sufficiently well-defined so as to permit the creation of an intermediary interoperability layer by which concepts can be translated from one information context to another without forcing rigid adherence to a specific technology stack. This is desirable for a number of reasons:

- *Interoperability future-proofs the proposed DM method against changes in technology*—arguably an inevitability in the volatile cloud computing domain. Individual components defined in the proposed architecture for adaptive containerised applications can be implemented differently from before without having to redesign the whole system.

- *Interoperability permits the existence of alternative implementations of components defined in the proposed architecture*—for example, two different versions of the Self-Adapter component might use different internal algorithms.

- *Interoperability allows components defined in the proposed architecture to be taken out and reused in other, new contexts*—hence increasing the payoff of development.

Therefore, there needs to be a set of minimal formal specifications for information objects used within the proposed auto-scaling method in this thesis. In other words, an abstract formal description of concepts is required with which these formal specifications can be associated. To this end, an ontology has to be defined in this thesis to unambiguously and formally specify the relevant notions of the proposed DM auto-scaling method. The following tables shows all classes and their own properties to be created in this ontology defined in the Knowledge Base.

A Service Cluster is a set of container instances providing a specific service to reply to the end-users' requests. Each Service Cluster offers an individual service. The Service Cluster class along with its own properties defined in the ontology is presented in Tab. 4.1.

Table 4.1

Service Cluster class

Property Name	Range (Type)	Rationale
Cluster_id	unsignedInt	This property allows a unique number to be generated when a new individual as a Service Cluster instance is inserted into the Knowledge Base.
Cluster_Name	string	This property implies the Service Cluster's name. It is usually defined as the service name.
Cluster_Owner	User_id	This property implies the Service Cluster's owner who offers cloud-based application providing the service to the end-users.
Cluster_SLA	SLA_id	Each Service Cluster has its own SLA (service level agreement) consisting of such as "End Time", "Start Time", etc.
Cluster_PortNumber	unsignedInt	A port serves as an endpoint in an operating system for many types of communications to provide multiplexing and identify services or processes.
Cluster_Metric	MonitoringMetric_id	Monitoring system at application level focuses on collecting data related to the application execution. Each metric has some properties such as id, name, data type (percentage, KBps, MBps, etc.), minimum, maximum, collecting period, and so on.
Cluster_Container	Container_id	Each Service Cluster includes one or more Container Instances providing the service to reply to the end-users' requests.

A Container Instance performs a specific task to run a service. Each Service Cluster includes one or more Container Instances providing the service to reply to the end-users' requests. The Container Instance class along with its own properties defined in the ontology is presented in Tab. 4.2.

Table 4.2

Container Instance class

Property Name	Range (Type)	Rationale
Container_id	unsignedInt	This property allows a unique number to be generated when a new individual as a Container Instance is inserted into the Knowledge Base.
Container_Image	Image_id	Each Container Instance is instantiated based on a Container Image stored on a local repository or Docker hub.
Container_PortNumber	unsignedInt	The service provided by the Container Instance uses a set of ports which need to be exposed.
Container_CPUProportion	decimal	Each Container Instance located on a host machine achieves a proportion of CPU cycles (assigned relative weight).
Container_MemoryLimit	decimal	Each Container Instance located on a host machine has its own limit at the use of memory.
Container_Host	Host_id	Each instantiated Container Instance is located on a Host Machine.
Container_Cluster	Cluster_id	Each Container Instance belongs to a Service Cluster to reply to the end-users' requests.

A Container Image consists of a pre-configured files and software. The purpose of a Container Image is to simplify delivery and operation of a containerised application instance. The Container Image class along with its own properties defined in the ontology is presented in Tab. 4.3.

A Host Machine is a system that not only exhibits the behaviour of a separate computer, but is also capable of performing tasks such as running container instances. The Host Machine class along with its own properties defined in the ontology is presented in Tab. 4.4.

A Monitoring Agent is a metric collector utilised to gather raw metrics and generate time-stamped monitoring events. It is installed in each Container Instance in the initial deployment. Monitoring Agents run in the background, therefore there is no interfering with the service' execution. The Monitoring Agent class along with its own properties defined in the ontology is presented in Tab. 4.5.

Table 4.3

Container Image class

Property Name	Range (Type)	Rationale
Image_id	unsignedInt	This property allows a unique number to be generated when a new individual as a Container Image is inserted into the Knowledge Base.
Image_Name	string	Each Container Image stored in a local repository or Docker hub has its own name.
Image_Description	string	This property is used to display the description about Container Image.
Image_Title	string	Title is a short and keywords-relevant description of Container Image.
Image_Version	string	During the cloud-based application lifecycle, Container Images may be upgraded several times.
Image_GenerationTime	dateTime	This property specifies when a particular Container Image was generated.
Image_FileFormat	string	This property is used to define the file format of Container Image.
Image_IRI	anyURI	Container Images need to be properly indexed. The semantic model will contain information about the geographic location, the URI (Uniform Resource Identifier), and other details for the search facility.
Image_Owner	string	This property implies the owner of Container Image that can be for example the application developer and the service provider.
Image_Functionality	string	This property implies Functionality which is provided by the service included in Container Image.

Table 4.4

Host Machine class

Property Name	Range (Type)	Rationale
Host_id	unsignedInt	This property allows a unique number to be generated when a new individual as a Host Machine instance is inserted into the Knowledge Base.
Host_Container	Container_id	Each Host Machine performs one or more Container Instances at the same time. All Container Instances running on a Host Machine provide different services.
Host_SLA	SLA_id	Each Host Machine has its own Service Level Agreement (SLA) consisting of such as "End Time", "Start Time" and so on.
Host_NetworkAdapter	string	Each Host Machine has its own network adapter to transmit and receive data such as etho.
Host_NetworkSpeed	unsignedInt	This property shows how much bandwidth is assigned to the Host Machine.
Host_IP	string	Each Host Machine has its own IP address to use the Internet Protocol for communication.
Host_SubnetMask	string	Each Host Machine has its own Subnet address in the network area.
Host_DefaultGateway	string	Default gateway is the node that is assumed to know how to forward packets on to other networks.
Host_OperatingSystem	OperatingSystem_id	Each Host Machine has an Operating System (OS).
Host_VirtualStorage	decimal	Each Host Machine has its own virtual storage with a particular storage size.
Host_CPUCount	unsignedInt	This property implies the number of processors which Host Machine has.
Host_CPUClockRate	decimal	This property implies the "Hertz" which is the measure of frequency in cycles per second.
Host_MemorySize	decimal	This is a parameter to define the size of an individual Host Machine's memory.

Table 4.5

Monitoring Agent class

Property Name	Range (Type)	Rationale
MonitoringAgent_id	unsignedInt	This property allows a unique number to be generated when a new individual as a Monitoring Agent instance is inserted into the knowledge base.
MonitoringAgent_Container	Container_id	Each Container Instance includes a Monitoring Agent. This property implies the Container Instance to which the Monitoring Agent belongs.
MonitoringAgent_Logging	boolean	When this property is set to true, the Monitoring Agent will log abnormal behaviours.
MonitoringAgent_ServerIP	string	Monitoring Agent uses this value defined in this property to determine the IP address of the Monitoring Server to which metrics will be distributed.
MonitoringAgent_DistributorPort	unsignedInt	This is a port which the Monitoring Agent uses to distribute metrics to Monitoring Server. It should be the same as the MonitoringServer_BindPort defined for the Monitoring Server instance.
MonitoringAgent_Metric	MonitoringMetric_id	Each Monitoring Agent measures different Monitoring Metrics which have some properties such as metric name, metric unit, minimum value, maximum value, etc.

A Monitoring Server receives measured value of Monitoring Metrics sent from Monitoring Agents and also it stores such values in the TSDB database. The Monitoring Server class along with its own properties defined in the ontology is presented in Tab. 4.6.

Table 4.6

Monitoring Server class

Property Name	Range (Type)	Rationale
MonitoringServer_id	unsignedInt	This property allows a unique number to be generated when a new individual as a Monitoring Server instance is inserted into the Knowledge Base.
MonitoringServer_IP	string	This property implies the Monitoring Server's IP.
MonitoringServer_Logging	boolean	When this property is set to true, the Monitoring Server will log abnormal behaviours.
MonitoringServer_BindIP	string	The network interface to which the Monitoring Server's Lister will bind. The default value is set to "*" indicating that it will bind to all network interfaces. If it must be changed then it is suggested to use the etho interface.
MonitoringServer_BindPort	unsignedInt	The port which Monitoring Server will bind to and listen for metric messages distributed by Monitoring Agents. This property should be the same as the MonitoringAgent_DistributorPort of each underlying Monitoring Agent.
MonitoringServer_DBHost	string	The IP address of the TSDB database backend. Default value defined for this property is localhost.
MonitoringServer_DBUser	string	The username for the TSDB database utilisation.
MonitoringServer_DBPass	string	The password for the TSDB database utilisation.
MonitoringServer_DB	string	The name of the TSDB database which will be used.
MonitoringServer_HeartPeriod	unsignedInt	The intensity in which the Monitoring Server HeartBeat should check for Monitoring Agent's availability.
MonitoringServer_HeartRetry	unsignedInt	The number of iterations that the Monitoring Server HeartBeat will allow a Monitoring Agent to be DOWN until it is declared as DEAD.

Monitoring Metrics are measured at different layers of the underlying cloud infrastructure, as well as possible performance parameters from deployed services. The Monitoring Metric class along with its own properties defined in the ontology is presented in Tab. 4.7.

Table 4.7

Monitoring Metric class

Property Name	Range (Type)	Rationale
MonitoringMetric_id	unsignedInt	This property allows a unique number to be generated when a new individual as a Monitoring Metric instance is inserted into the Knowledge Base.
MonitoringMetric_Name	string	This property implies the name of Monitoring Metric.
MonitoringMetric_Description	string	This is a longer, freestyle textual description of the Monitoring Metric.
MonitoringMetric_Group	string	For example, all metrics such as memTotal, memFree, memUsed and memUsedPerecent are belonging to a group named "Memory".
MonitoringMetric_Level	unsignedInt	Important Monitoring Metrics can be monitored in IaaS and in SaaS scenarios which are respectively infrastructure-level and application-level.
MonitoringMetric_Unit	string	Monitoring Metric's unit could be percentage (%), KBps, MBps, Bps, Yes/No and so on.
MonitoringMetric_DataType	string	Monitoring Metric's data type could be integer, double, etc.
MonitoringMetric_CollectingInterval	unsignedInt	The collecting period is an important parameter for the time-critical environment. It means monitoring frequency, the interval between each measurement for the metric.
MonitoringMetric_History	unsignedInt	How many days the monitoring system keeps the collected data.
MonitoringMetric_UperLimit	decimal	This property implies the maximum value of the metric that could be observed.
MonitoringMetric_LowerLimit	decimal	This property implies the minimum value of the metric that could be observed.
MonitoringMetric_Threshold	decimal	This property implies the threshold value of the metric that should be continuously checked.
MonitoringMetric_Agent	MonitoringAgent_id	Each Monitoring Metric instance is measured by a Monitoring Agent.

Dynamically specifying Auto-scaling Rules provides for fine-grained reaction to workload fluctuations, and thus it can improve application performance and a higher level of resource utilisation. The Auto-scaling Rule class along with its own properties defined in the ontology is presented in Tab. 4.8.

The Alarm-trigger component is able to alert if any threshold associated with a Monitoring Metric is violated. The Alarm Notification class along with its own properties defined in the ontology is presented in Tab. 4.9.

Table 4.8

Auto-scaling Rule class

Property Name	Range (Type)	Rationale
Rule_id	unsignedInt	This property allows a unique number to be generated when a new individual as an Auto-scaling Rule is inserted into the Knowledge Base.
Rule_Date	dateTime	This property specifies the date when a particular Auto-scaling Rule is generated.
Rule_Time	dateTime	This property specifies the time when a particular Auto-scaling Rule is generated.
Rule_ThroughputT0	decimal	This property implies the value of application throughput in the current interval.
Rule_ThroughputT1	decimal	This property implies the value of application throughput in the previous interval.
Rule_ThroughputT2	decimal	This property implies the value of application throughput in the second last interval.
Rule_ResponseTime	decimal	This property implies the current value of response time provided.
Rule_CPU	decimal	This property implies the current value of average CPU utilisation of the whole cluster
Rule_Memory	decimal	This property implies the current value of average memory utilisation of the whole cluster
Rule_ClusterSize	unsignedInt	This property implies the current value of cluster size which is the current number of container instances in the cluster.
Rule_Action	Integer	This property implies the adaptation action which is the number of container instances to be instantiated or terminated.

Table 4.9

Alarm Notification class

Property Name	Range (Type)	Rationale
Notification_id	unsignedInt	This property allows a unique number to be generated when a new individual as an Alarm Notification is inserted into the Knowledge Base.
Notification_Date	dateTime	This property specifies the date when a particular Alarm Notification is generated.
Notification_Time	dateTime	This property specifies the time when a particular Alarm Notification is generated.
Notification_Metric	MonitoringMetric_id	Each Alarm Notification belongs to a specific Monitoring Metric which has some properties such as metric name, metric unit, minimum value, maximum value, etc.
Notification_Cluster	Cluster_id	Each Alarm Notification belongs to a specific Service Cluster.
Notification_Value	decimal	This property implies the current value of the metric which violates the Monitoring Metric's threshold.
Notification_Type	string	This property implies the message type of Alarm Notification that can be "warning" or "critical".

4.7 *Alarm-Trigger*

The Alarm-Trigger component is a rule-based entity which continuously checks the incoming monitoring data and notifies the Self-Adapter component when the system is going to experience abnormal situation. The Alarm-Trigger component constantly processes two functions. A function named Investigating for Container Instantiation (*IFCI*) is defined in the Alarm-Trigger component to check if it is required to launch new container instances. Furthermore, another function named Investigating for Container Termination (*IFCT*) is defined in the Alarm-Trigger component in order to evaluate if one of the running containers can be stopped without any application performance deterioration.

A significant application-level metric which is adopted in the operation of the Alarm-Trigger component is the service response time. In this regard, the threshold for the service response time (T_{res}) should be set. In order to make the system prevent any performance degradation, the value of T_{res} needs to be set more than the usual time period to perform a single job without any problem when the system is not overloaded. In the case that T_{res} is set very close to the value of the usual time to process a single job, the auto-scaling method may lead to unnecessary changes in the number of running containers in the cluster, whereas the system is currently capable of providing end-users a suitable performance without any threat. Moreover, if T_{res} is set too much bigger than the value of the usual time to perform a single job, the auto-scaling approach may be less sensitive to application performance and more dependent on infrastructure usage.

In many cloud resource management systems [129–134], the primitive thresholds for the CPU and memory utilisation (T_{cpu} and T_{mem}) are set to the value of 80%. If the value of these two primitive thresholds is set closer to 100%, therefore the auto-scaling method has no chance to timely react to runtime changes in the workload density before a performance degradation arises. If the value of these two primitive thresholds used by the Alarm-Trigger is set less than 80%, then this may cause an over-provisioning issue which wastes costly resources. In the execution environment, if the workload trend is very even and predictable, these two primitive thresholds for the utilisation of CPU and memory can be pushed higher than 80%.

According to *IFCI* shown in Fig. 4.5, if one of the average CPU or memory utilisation of the cluster (*AvgCpu* or *AvgMem*) surpasses the associated threshold (T_{cpu} or T_{mem}, 80%) and the average response time (*AvgRT*) is more than T_{res}, the Self-Adapter will be notified since the number of container instances in the cluster should increase on demand.

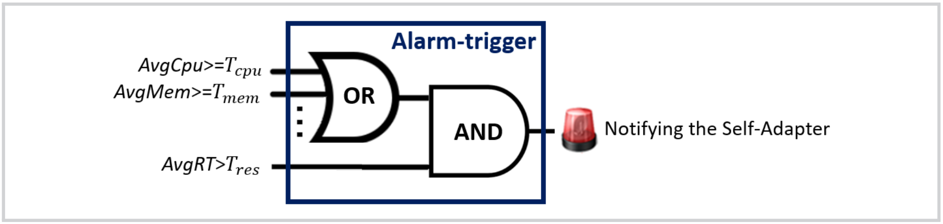


Figure 4.5
Upscaling principle defined in the Alarm-Trigger.

Involving the average response time in *IFCI* shown in Fig. 4.6 tends to prevent 20% (100-T_{cpu} or 100-T_{mem}) resources waste. In other words, there is the possibility that the system may work at even 100% resource utilisation without instantiating more new containers. This is because the average response time is appropriately satisfying. Or in essence, the average response time is below T_{res}. In *IFCI*, *cpu_usage_of_container* and *memory_usage_of_container* which are numbered from 1 to N are the CPU and memory utilisation of each individual container instance in the cluster. As an illustration, *cpu_usage_of_container1* is the CPU utilisation of the first container instance, *cpu_usage_of_container2* is the CPU utilisation of the second container instance, and so on. AVG_Cluster is the grouping operator employed to calculate the average CPU and memory utilisation of the cluster that are nominated as *AvgCpu* and *AvgMem* respectively.

```
Inputs:
$T_{cpu}$: Threshold for the average CPU usage of the cluster
$T_{mem}$: Threshold for the average memory usage of the cluster
$T_{res}$: Threshold for the average response time
Outputs:
If it is needed to notify the Self-Adapter in order to prevent under-provisioning
Cluster = [Container1,..., ContainerN]
AvgCpu=AVG_Cluster(cpu_usage_of_container1,...,cpu_usage_of_containerN);
AvgMem=AVG_Cluster(memory_usage_of_container1,..., memory_usage_of_containerN);
if (((AvgCpu>=$T_{cpu}$) or (AvgMem>=$T_{mem}$)) and (AvgRT>$T_{res}$)) then
call ContainerInitiation(); // call CI() to start new containers
```

Figure 4.6

IFCI defined in Alarm-Trigger.

IFCT (shown in Fig. 4.7) has been defined to investigate the feasibility of decreasing the number of running containers in the cluster, without any QoS degradation perceived by end-users. In order to enhance the stability of the system and to ensure that the system provides an appropriate service quality to users, it is presumed that if a container instance is launched and added to the cluster, any container termination should not occur during the next two adaptation intervals, even if the average CPU or memory usage of the cluster is quite low.

In some situations, the workload density instantly drops down only for a short period of time; whereas after that, the number of requests will increase again very soon. In such workload scenarios, avoiding any change in the cluster size (number of container instances running in the service cluster) is beneficial to reach a greater level of opera-

```
Inputs:
$T_{cpu}$: Threshold for the average CPU usage of the cluster
$T_{mem}$: Threshold for the average memory usage of the cluster
Outputs:
If it is needed to notify the Self-Adapter in order to prevent over-provisioning
Cluster = [Container1,..., ContainerN]
AvgCpu=AVG_Cluster(cpu_usage_of_container1,...,cpu_usage_of_containerN);
AvgMem=AVG_Cluster(memory_usage_of_container1,..., memory_usage_of_containerN);
if (((AvgCpu<$T_{cpu}$) or (AvgMem<$T_{mem}$)) and (no container addition in the last 2 intervals)) then
call ContainerTermination(); // call CT() to stop one of containers if possible
```

Figure 4.7

IFCT defined in Alarm-Trigger.

tional environment's stability. Therefore, the auto-scaling method should wait at least one adaptation interval after any container instantiation to make sure there is not a short drop in the workload. This time interval after any container instantiation is called waiting period shown in Fig. 4.8.

After waiting period, in order to consider any possible container termination, the auto-scaling method observes the status of execution environment during one whole adaptation interval without any container instantiation. This time interval is called observing period shown in Fig. 4.8.

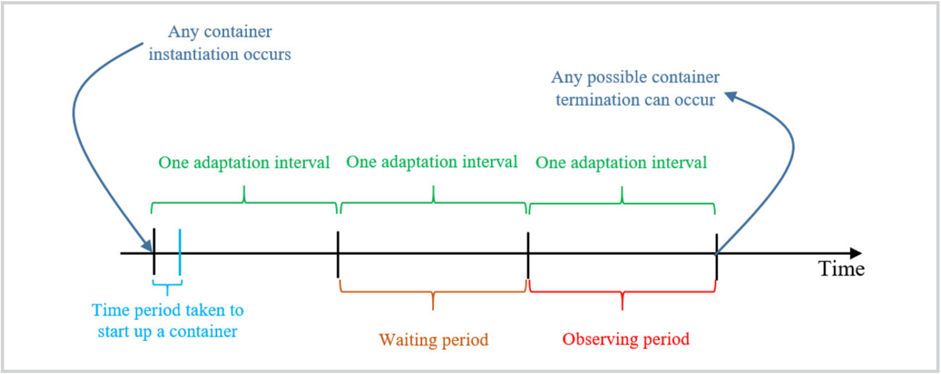

Any container instantiation occurs
Any possible container termination can occur
One adaptation interval
One adaptation interval
One adaptation interval
Time
Time period taken to start up a container
Waiting period
Observing period

Figure 4.8

Waiting period and observing period considered by Alarm-Trigger.

Therefore, any possible container termination may happen if there is no container instantiation during the last two intervals (the waiting period and the observing period), as demonstrated in Fig. 4.7.

The Alarm-Trigger component is capable of fetching a YAML file including all the inputs mentioned in two functions called *IFCI* and *IFCT*. This YAML file is exposed by the Web-based IDE via a prepared API. Instructions for the usage of the implemented Alarm-Trigger component are described at GitHub [135] published under the Apache 2 license.

4.8 Self-Adapter

When predefined conditions are met (e.g. thresholds are violated), the Self-Adapter component is called by the Alarm-Trigger component. Widely used static rule-based auto-scaling approaches are adopted in the current production systems which mainly employ fixed, single-level rules. In contrast to such approaches, the proposed DM auto-scaling method uses both container-level metrics (e.g. CPU, memory, etc.) as well as application-level metrics (e.g. response time and application throughput) which can dynamically influence the adjustable auto-scaling rules. In other words, in our proposed DM auto-scaling method, conditions when container instances are instantiated or stopped are dynamic and are not predefined since it strongly depends on many factors at runtime as shown in Fig. 4.9.

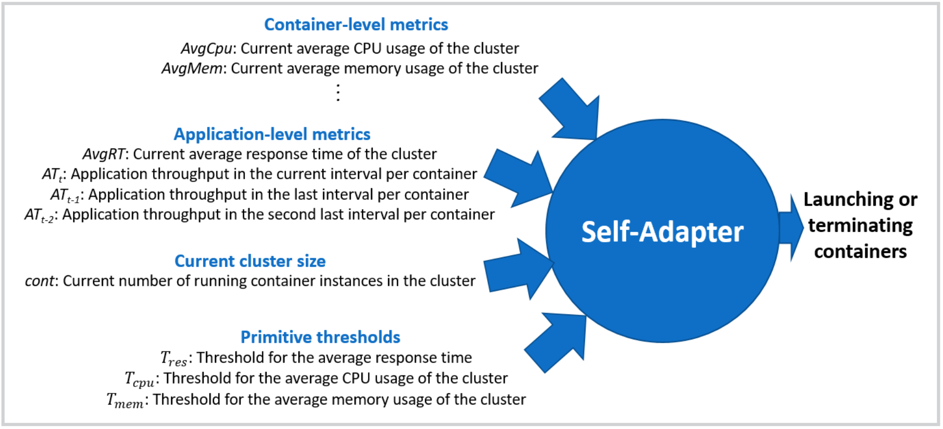

Container-level metrics
AvgCpu: Current average CPU usage of the cluster
AvgMem: Current average memory usage of the cluster
⋮
Application-level metrics
AvgRT: Current average response time of the cluster
AT_t: Application throughput in the current interval per container
AT_{t-1}: Application throughput in the last interval per container
AT_{t-2}: Application throughput in the second last interval per container
Current cluster size
cont: Current number of running container instances in the cluster
Primitive thresholds
T_{res}: Threshold for the average response time
T_{cpu}: Threshold for the average CPU usage of the cluster
T_{mem}: Threshold for the average memory usage of the cluster
Self-Adapter
Launching or terminating containers

Figure 4.9
Inputs of the Self-Adapter component.

The Self-Adapter includes two functions which are in charge of proposing adaptation actions. One function called Container Instantiation (*CI*) is to initiate new containers to enhance the performance of the application. The principle of *CI* function defined in the Self-Adapter is shown in Fig. 4.10.

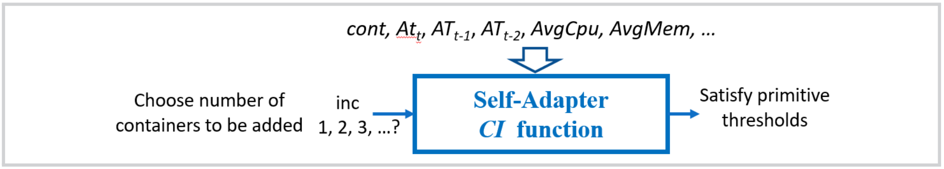

cont, At_t, AT_t-1, AT_t-2, AvgCpu, AvgMem, …
Choose number of containers to be added
inc 1, 2, 3, …?
Self-Adapter CI function
Satisfy primitive thresholds

Figure 4.10

Upscaling principle defined in the Self-Adapter.

As shown in Fig. 4.10, according to a set of infrastructure-level and application-level metrics, the proposed *CI* function chooses the minimum number of containers to be added to the cluster in order to satisfy primitive thresholds. The pseudocode of the *CI* function, defined in the Self-Adapter, is illustrated in Fig. 4.11.

```
Inputs:
$T_{cpu}$: Threshold for the average CPU usage of the cluster
$T_{mem}$: Threshold for the average memory usage of the cluster
AvgCpu: Current average CPU usage of the cluster
AvgMem: Current average memory usage of the cluster
$AT_t$: Application throughput in the current interval per container
$AT_{t-1}$: Application throughput in the last interval per container
$AT_{t-2}$: Application throughput in the second last interval per container
cont: Current number of running container instances in the cluster
Outputs:
Launching new container instance(s)
-----
inc1 ← 0;
if ($AvgCpu > T_{cpu}$) then {
    do {
        inc1++;
        $P_{cpu} \leftarrow (\frac{cont \times AvgCpu \times [(2\times\frac{ATt}{ATt-1}+\frac{ATt-1}{ATt-2})/3]}{cont+inc1})$;
    } while ($P_{cpu} > T_{cpu}$);
} // end of if
inc2 ← 0;
if ($AvgMem > T_{mem}$) then {
    do {
        inc2++;
        $P_{mem} \leftarrow (\frac{cont \times AvgMem \times [(2\times\frac{ATt}{ATt-1}+\frac{ATt-1}{ATt-2})/3]}{cont+inc2})$;
    } while ($P_{mem} > T_{mem}$);
} // end of if
inc ← max(inc1, inc2);
initiate_new_containers(inc); // start 'inc' new container(s)
```

Figure 4.11

CI defined in Self-Adapter.

When *CI* function is called, it starts predicting the average CPU and memory utilisation of the cluster with respect to "current number of container instances in the cluster," "current average resource usage of the cluster," and "the amount of increase in the rate of throughput" if one or more new container would be launched and added to the cluster. Based on these two predicted values (P_{cpu} and P_{mem}) for the average CPU and memory utilisation of the cluster, the number of new container instances which need to be added to the cluster is determined.

CI function (shown in Fig. 4.11) defined in the Self-Adapter component needs to estimate the increasing rate of throughput in the next interval. In this regard, three possible formulas can be used, as Formula (A), Formula (B) and Formula (C).

Formula (A) $\left(\frac{AT_t}{AT_{t-1}}\right)$

Formula (B) $[\left(2 \times \frac{AT_t}{AT_{t-1}} + \frac{AT_{t-1}}{AT_{t-2}}\right) \div 3]$

Formula (C) $[\left(3 \times \frac{AT_t}{AT_{t-1}} + 2 \times \frac{AT_{t-1}}{AT_{t-2}} + \frac{AT_{t-2}}{AT_{t-3}}\right) \div 6]$

In order to estimate the increasing rate of throughput in the next interval, Formula (A) considers only the rate of throughput in the current interval; whereas Formula (B) takes into consideration not only the rate of throughput in the current interval $\left(\frac{AT_t}{AT_{t-1}}\right)$, but also the rate of throughput in the previous interval $\left(\frac{AT_{t-1}}{AT_{t-2}}\right)$. However, in order to estimate the increasing rate of throughput in the next interval, the importance of throughput rate in the current interval is more than the importance of throughput rate in the previous interval. This is because time intervals are consecutive, and the throughput in the next interval probably would be closer to the throughput in the last interval than the throughput in the second last interval. Consequently, the weight assigned to the throughput rate in the current interval has been set to 2 which is more than the weight assigned to the throughput rate in the previous interval. Along the same line, Formula (C) takes into account also the throughput rate achieved before the previous interval $\left(\frac{AT_{t-2}}{AT_{t-3}}\right)$.

In order to choose the best formula to be used by the Self-Adapter to estimate the increasing rate of throughput in the next interval, we analysed a set of experiments performed based on the experimental setup which is explained in Chapter Results. To this end, a drastic increase (from 100 requests to 500 requests) in the workload density was examined. Fig. 4.12 shows the number of new container instances allocated by three different auto-scaling methods using Formula (A), Formula (B) and Formula (C).

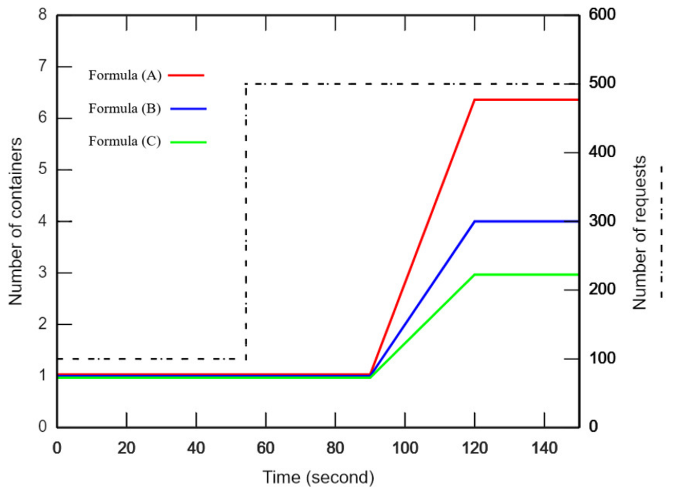


Figure 4.12

Number of new containers allocated by three different auto-scaling methods in response to a drastic increase in the workload.

Fig. 4.13 shows the response time provided by three different auto-scaling methods using Formula (A), Formula (B) and Formula (C).

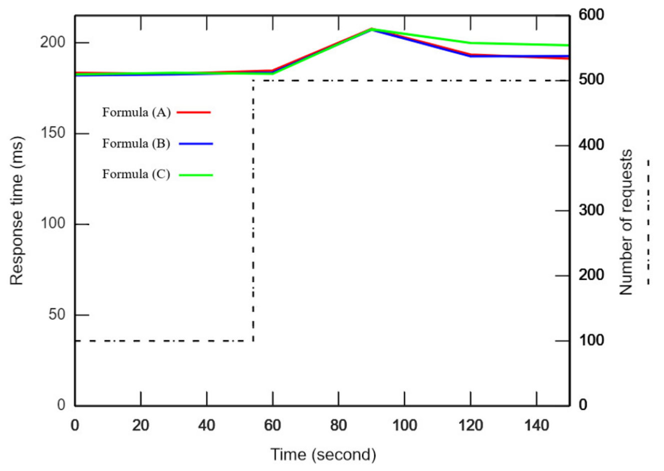


Figure 4.13

Response time provided by three different auto-scaling methods in response to a drastic increase in the workload.

When the workload density suddenly began to increase, the auto-scaling method which exploits Formula (A) allocated more container instances than other two methods. However, both auto-scaling methods using Formula (A) and Formula (B) offer almost the same response time, while allocating different number of new container instances. Therefore, if the Self-Adapter which uses Formula (A) takes into account only the rate of throughput in the current interval, the number of new instantiated containers will be more than what is needed when the system encounters a drastic increase in the workload density. In order to prevent such resource waste, the Self-Adapter component should consider not only the rate of throughput in the current interval, but also the rate of throughput in the previous interval, as described in Formula (B). Moreover, the Self-Adapter which employs Formula (C) allocated less number of containers in comparison with other two approaches. Hence, the response time provided by this auto-scaling method using Formula (C) was inappropriately affected by the drastic increase in the workload. This is because considering the throughput rate observed before the previous interval can reduce the agility of the proposed auto-scaling method in order to recognise the drastic increase which appears in the workload. As a consequence, Formula (B) as the best option was used in *CI* function defined in the Self-Adapter to estimate the increasing rate of throughput in the next interval, demonstrated in Fig. 4.11.

If more than one container is required to be instantiated, the Self-Adapter component runs all needed container instances concurrently. Accordingly, the adaptation interval, which is the minimum period when the next adaptation event occurs, has to be specified longer than the start-up time of a container instance. In such manner, if any auto-scaling action happens, the whole system can continue operating properly without losing control over containers running in the cluster.

Another function called Container Termination (*CT*) is responsible for possibly terminating unnecessary containers to prevent resource over-provisioning. Here we describe how the termination of non-required container instances for a CPU-intensive application takes place. Let us assume that the cluster size or the number of container instances in the cluster is two. If the average CPU usage of the cluster which consists of these two running container instances is less than $\left(\frac{T_{cpu}}{2}\right) - \alpha$, one of these two containers should be stopped. In this formula, the constant α can have a value between 0% and 10%, that helps the proposed DM auto-scaling method conservatively make sure that the container termination will not cause an unstable situation.

Experimenting with equal computational requirements and workload density, an up to 10% variation in the average CPU and memory utilisation of the cluster (*AvgCpu* and *AvgMem*) can still be seen. This difference is the effect of runtime variations in running conditions that are out of the application providers' control. With respect to this rationale, the maximum value for α can be set to the value of 10%.

It should be noted that a value of α can closer to 0% may be unsuccessful in providing the essential robustness of auto-scaling methodology. Since due to minor variations in the average CPU usage of around $\left(\frac{T_{cpu}}{2}\right)$, the system may terminate a container at that moment, and afterwards shortly would launch a new one again. A value α closer to 10% may reduce the efficiency of the proposed adaptation method because, in such a case, redundant containers have less possibility of being terminated from the cluster in general. Therefore, a higher value of α would possibly cause longer periods of over-provisioned resources. For the experimentation in this thesis, the value of α has been set to 5%, which results in neither unnecessary over-provisioning of resources, nor too frequent changes in the number of running container instances.

In order to achieve a stable operational environment while increasing the resource utilisation, α used by the Self-Adapter component is aimed at avoiding frequent changes in the cluster size due to minor fluctuations in the workload density. Fig. 4.14 shows how this constant tries to sustain the number of running containers providing the service at the same level when fluctuations in the varied number of requests are not severe at run-time. During the time period highlighted in blue, the constant α provides the expected robustness of auto-scaling method while there exists a trembling workload which does not change drastically. Afterwards, when the workload density drops more, a running container instance is terminated to improve the resource utilisation without any application performance degradation.

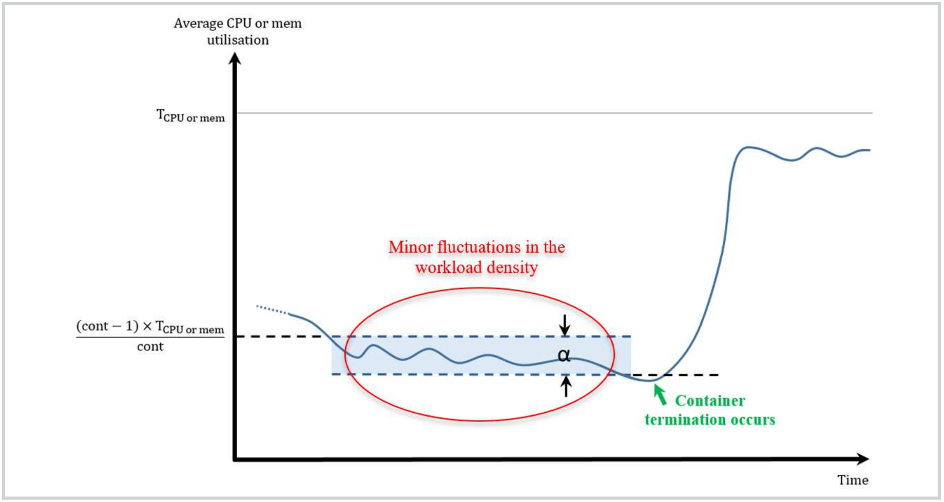

Average CPU or mem utilisation
T_CPU or mem
(cont − 1) × T_CPU or mem / cont
Minor fluctuations in the workload density
α
Container termination occurs
Time

Figure 4.14

Avoiding frequent changes in the cluster size due to minor fluctuations in the workload density.

Consequently, assume that two container instances are running in the cluster, if the average CPU utilisation of the cluster is less than $\left(\frac{80}{2}\right)$-5=35 percent, it is possible to terminate one of the running container instances. The reason is that with the current workload density after stopping the container, the average CPU usage of the cluster would be at most 70%, which is less than T_{cpu} at 80%. Along the same line, given that if three container instances are running in the cluster and the average CPU utilisation of the cluster is under $\left(\frac{(3-1)*T_{cpu}}{3}\right) - \alpha$, one of the container instances could be terminated, since in this manner there would not be any degradation in the QoS of the application.

Generally, it was assumed if the current number of container instances running in the cluster is *cont*, and the average CPU usage of the cluster is below β_{cpu} specified by Eq. (4.1), it is possible to stop one of the container instances running in the cluster without compromising the application performance. Besides that, for a memory-bound service, β_{mem}, which is thoroughly similar to β_{cpu}, allows to determine the possibility of reducing the number of running containers in the cluster if required upon the memory utilisation, as Eq. (4.2).

$$\beta_{cpu} = \left(\frac{(cont-1) \times T_{cpu}}{cont}\right) - \alpha \tag{4.1}$$

$$\beta_{mem} = \left(\frac{(cont-1) \times T_{mem}}{cont}\right) - \alpha \tag{4.2}$$

Therefore, the threshold (β_{cpu} or β_{mem}) to terminate one of the running container instances in the cluster dynamically depends on different parameters at runtime: the primitive thresholds for the utilisation of CPU and memory (T_{cpu} and T_{mem}), the constant α and the current number of running containers in the cluster (*cont*), as shown in Fig. 4.15.

Fig. 4.16 presents the pseudocode of the *CT* function, proposed to be defined in the Self-Adapter component, called by the Alarm-Trigger. Based on the average CPU and memory utilisation of the cluster and other inputs such as T_{cpu}, T_{mem}, *cont* and α can, this function specifies if it is needed to reduce the number of running container instances in the cluster.

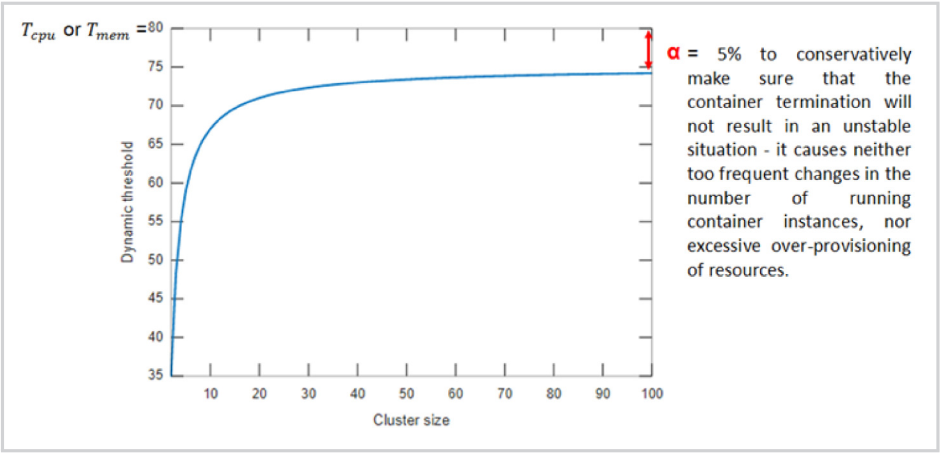


Figure 4.15

Downscaling principle defined in the Self-Adapter.

```
Inputs:
AvgCpu: Current average CPU usage of the cluster
AvgMem: Current average memory usage of the cluster
$T_{cpu}$: Threshold for the average CPU usage of the cluster
$T_{mem}$: Threshold for the average memory usage of the cluster
cont: Current number of running container instances in the cluster
α: Conservative constant to avoid an unstable situation
Outputs:
Terminating an unnecessary container instance if it is possible
dec1 ← 0;
$\beta_{CPU}$ ← Calculate($T_{cpu}$, cont, α);
if (AvgCpu<$\beta_{CPU}$) then dec1 ← 1;
dec2 ← 0;
$\beta_{mem}$ ← Calculate($T_{mem}$, cont, α);
if (AvgMem<$\beta_{mem}$) then dec2 ← 1;
dec ← min(dec1, dec2);
if (dec==1) then terminate_one_container(); // Stop one container
```

Figure 4.16

CT defined in Self-Adapter.

The proposed auto-scaling method ensures the QoS of the application by stopping at most one container instance in each adaptation interval. In such a manner, after any termination of a container, the *CT* function assuredly provides acceptable service responses within continuously changing, uncertain environments at runtime. For example, this auto-scaling strategy can be adopted to handle on-off workload patterns in which peak spikes appear periodically in short time intervals. An instance of an on-off workload pattern is depicted in Fig. 4.17.

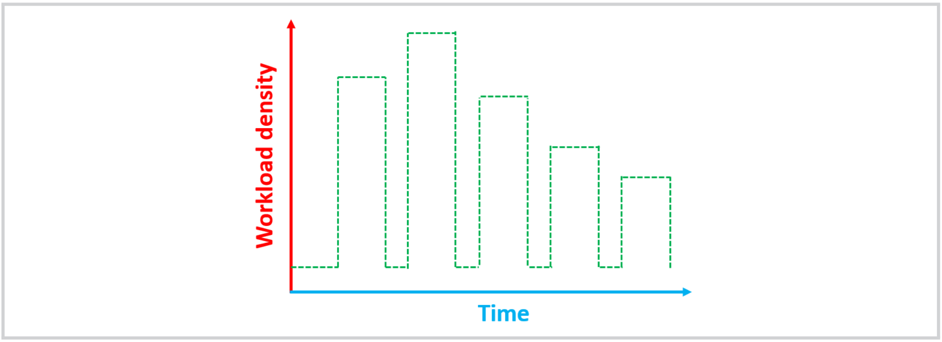


Figure 4.17

An example for on-off workload pattern.

In such workload patterns, shown in Fig. 4.17, stopping most of the container instances running in the cluster at once when the number of requests instantly drops down a lot is not a suitable adaptation action. This is because more containers running into the pool of resources will be required very soon. This non-conservative strategy may cause too many container terminations and instantiations with the consequent QoS degradation. It means that the shutdown and start-up times of container instances need to be taken into consideration during these types of on/off workload patterns.

4.9 Using the DM method as a third party tool

In order to use the proposed DM auto-scaling method as a third party tool, the following subsections should be followed, respectively.

Using the Monitoring Server, TSDB and web-based IDE: The developed monitoring system is freely accessible to both researchers and commercial companies on GitHub [41] under an Apache 2 license. A container image which can be easily instantiated has been already built to include three different entities: (i) Monitoring Server, (ii) TSDB and (iii) the web-based IDE. This container image is open-source and publically released on Docker Hub [81]. It takes almost one minute to run the Monitoring Server. The Monitoring Server should be running on a machine with enough memory, disk and CPU resources. This machine should address the Cassandra hardware requirements explained in the following page:

`https://wiki.apache.org/cassandra/CassandraHardware`

Using the Monitoring Agent: According to the proposed auto-scaling architecture for adaptive container-based applications, each container consists of two entities: (i) an application instance and (ii) a Monitoring Agent. The Monitoring Agent is the actual component which collects individual metrics' values. The development of the Monitoring Agent can follow three different approaches. Some applications such as Apache Tomcat application server provide APIs for obtaining metrics value. In this situation, the Monitoring Agent internally queries this API and retrieves the values. On the other hand, some applications such as Jena Fuseki provide only access to log files. The Monitoring Agent therefore needs to parse the log file and obtain the metric values. For the applications that provide neither API nor log file, the application's source code needs to be modified in order to implement the Monitoring Agent.

To make the Monitoring Agent, the StatsD protocol is used in this work. Therefore, Monitoring Agents have to be implemented through StatsD protocol available for many programming languages such as C/C++ and Python. There are too many examples on the Web to make a StatsD client:

`github.com/etsy/statsd/wiki#client-implementations`

Using the Alarm-Trigger: Instructions for the use of the developed Alarm-Trigger component are explained at GitHub [135] published under the Apache 2 license. In order to use the proposed Alarm-Trigger, a container which has been already built needs to be simply instantiated. The Alarm-Trigger container image is open-source and publically released on Docker Hub [136]. It should be noted that before running the containerised Alarm-Trigger, two APIs should be prepared and accessible. The first API should expose a YAML file which determines the inputs such as metrics to be checked, their thresholds, their ranges of value, their checking periods, and so on. At the beginning, the Alarm-Trigger component will fetch this input and then start working. The template for the YAML file can be seen here:

`github.com/salmant/ASAP/blob/master/SWITCH-Alarm-Trigger`

Using the Self-Adapter: It should be noted that host machines allocated by the Self-Adapter to run containers should include the Docker engine on which the Docker's Remote API needs to be enabled. The implementation of the proposed Self-Adapter component is kept publically available at GitHub [137] published under the Apache 2 license:

`github.com/salmant/ASAP/blob/master/SWITCH-Self-Adapter`

5

Results

In this chapter, our proposed method called the Dynamic Multi-level (DM) auto-scaling mechanism is compared experimentally with seven other rule-based provisioning approaches. Similar to our method, all investigated methods are also rule-based and are currently considered as advanced auto-scaling approaches which are used in production systems. The results are presented with respect to various experiments which are designed and performed as a set of proof-of-concept implementations. The experiments will then be analysed using different statistical methods and tools. To this end, we evaluate the results with regards to different workload scenarios in order to check the application performance provided by our proposed auto-scaling method and the amount of resources allocated during the experiments.

5.1 Experimental design

The *httperf* [138] tool has been employed in order to build a load generator which is able to produce different workload patterns for various analyses in our empirical evaluation. In this regard, five various workload scenarios have been inspected, as shown in Fig. 5.1.

Each workload scenario investigated in this thesis implies various type of applications. A slowly rising/falling workload scenario may represent incoming traffic sent to an e-learning system in which daytime involves more task requests than at night. A drastically changing scenario may imply a heavy workload to be accomplished by a broadcasting news channel in which some news or a video rapidly spreads in the social media environment. This type of application commonly has a short active period, after which the provided service can be offered at the lowest service level. Systems such as batch processing systems process workload patterns similar to the on-off workload scenario in which task requests tend to be accumulated around batch runs regularly over short periods of time. A gently shaking workload scenario shows predictable execution environments such as household settings that enable application providers to define detailed requirements, and hence assign the accurate amount of resources to the system.

In this thesis, our proposed DM auto-scaling method has been compared with different rule-based provisioning approaches described in Section 3.2. These policies consist of HPA (Horizontal Pod Auto-scaling), TTS1 (1st method of Target Tracking Scaling), TTS2 (2nd method of Target Tracking Scaling), SS1 (1st method of Step Scaling), SS2 (2nd method of Step Scaling), THRES1 (1st method of THRESHOLD), and THRES2 (2nd method of THRESHOLD). The implementation of all these seven auto-scaling approaches as well as experimental data are kept publically available at our GitHub ac-

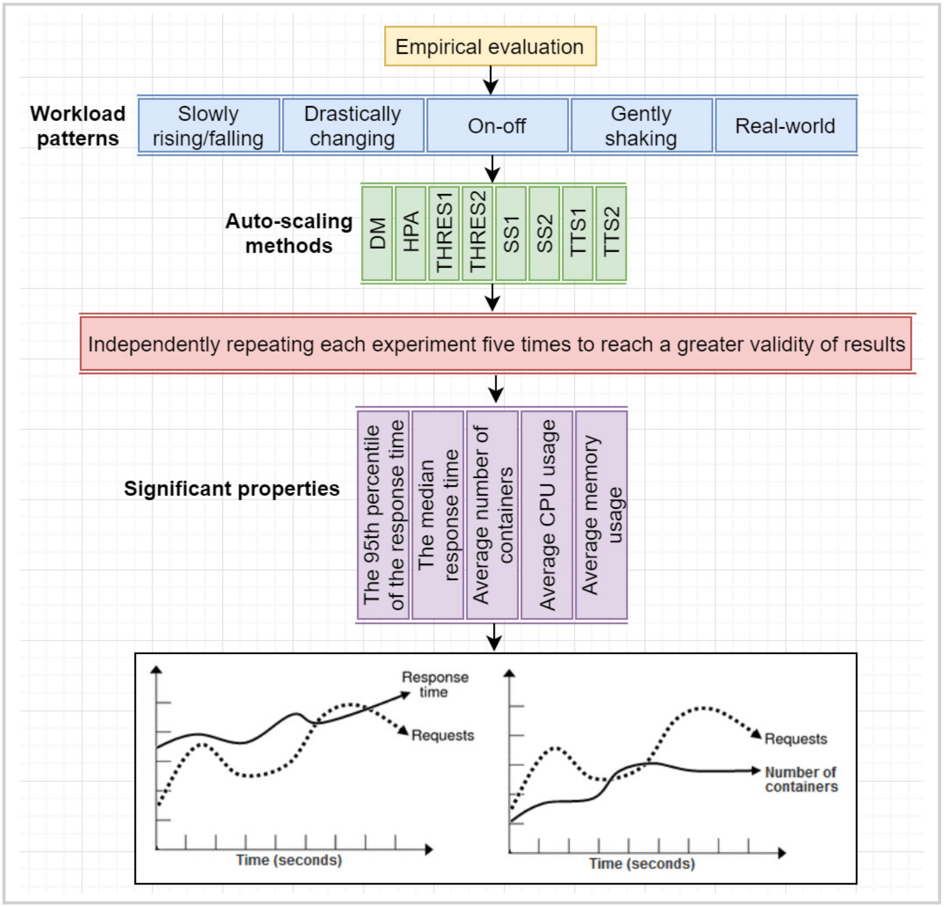


Figure 5.1

Experiment design to compare the new DM method to existing auto-scaling methods.

count [137]. Although, we did not implement Multiple Policies (MP) approach, as this provisioning method is not revealed clearly in terms of technical feasibility by the Google Cloud Platform.

Each experiment was repeated for five iterations to achieve the average values of important properties and to verify the obtained results and thus to have a greater validity of results. Accordingly, the results reported in this thesis are mean values over five runs for each experiment.

In every experiment, the results are analysed to ensure that examined auto-scaling methods are able to meet the core objectives from the perspective of application performance requirements (e.g. response time), while optimising the resource allocation in terms of the number of container instances. In this context, each auto-scaling approach is investigated primarily according to the 95th percentile of the response time, the median response time, average number of containers, and average CPU and memory usage.

5.2 Experimental setup

A set of experiments was performed to evaluate the choices of mentioned parameters (adaptation interval, T_{res} and α), and the sensitivity of the DM auto-scaling method to changes in the value of these parameters was analysed as follows:

Adaptation interval: The adaptation interval is the minimum duration between two successive adaptation actions. The adaptation interval should be defined longer than the time period taken to start up a container instance. This is because the auto-scaling method needs to make adaptation decisions when the system is quite stable. Our practical experiments indicate that the period of time taken to start up a container instance used in this thesis is around six seconds.

In order to choose the best time period for the adaptation interval to be used by the Self-Adapter, we performed a set of experiments according to three different time length: 15 seconds, 30 seconds and 60 seconds. To this end, a workload scenario has been inspected that includes a slowly rising workload, an off workload, a drastically increasing workload and a slowly falling workload, respectively by passage of time. Fig. 5.2 shows the number of new container instances allocated by three different auto-scaling methods using 15-second adaptation interval, 30-second adaptation interval and 60-second adaptation interval.

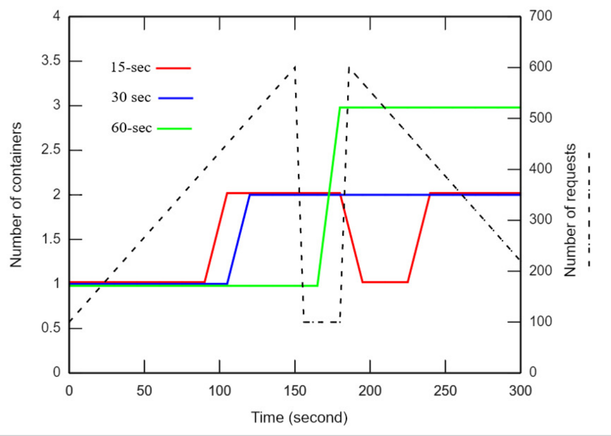


Figure 5.2

Number of containers allocated by three different auto-scaling methods using 15-second adaptation interval, 30-second adaptation interval and 60-second adaptation interval.

Fig. 5.2 demonstrates that the auto-scaling approach using 15-second adaptation interval is the fastest method since it was able to allocated a new container instance sooner than other methods in response to slowly rising workload from 100 to 600 requests. It also terminated the container added to the cluster when the workload density suddenly decreases in the off period. However, this adaptation action is not appropriate since there is the upcoming drastically increasing workload from 100 to suddenly 600 requests. That is why the response time provided by this method was slow in such a situation, shown in Fig. 5.3.

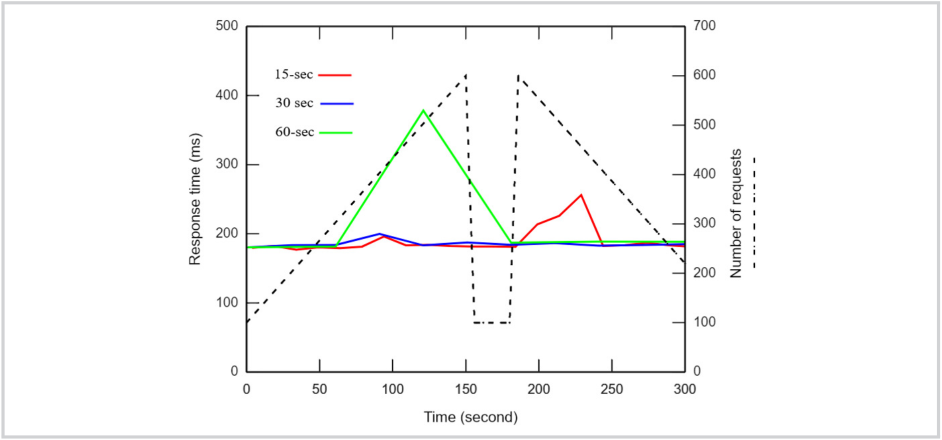


Figure 5.3

Response time provided by three different auto-scaling methods using 15-second adaptation interval, 30-second adaptation interval and 60-second adaptation interval.

Fig. 5.3 shows that the slowest response time offered by the auto-scaling approach using 60-second adaptation interval since it was not agile enough to recognise runtime changes in the workload density over time. However, this method employed more container instances (1.86) during the experiment compared with auto-scaling approaches using 15-second adaptation interval (1.52) and 30-second adaptation interval (1.62). The auto-scaling approach using 30-second adaptation interval was the method which provided the fastest response time almost steady over time (186.2 ms) on average in comparison with auto-scaling approaches using 15-second adaptation interval (194.28 ms) and 30-second adaptation interval (229.56 ms). Therefore in the experiments, the adaptation interval, which means the time period between two possible consecutive adaptation actions (increasing or reducing the cluster size), was specified as 30 seconds to ensure that there would be no issue if any auto-scaling event takes place. While the monitoring interval may be defined as very short in milliseconds, it was also set to 30 seconds to decrease the communication traffic load and any monitoring overhead for the measurements.

Threshold for the service response time (T_{res}): A finite element analysis application that is useful to solve mathematical physics and engineering problems has been developed and containerised to be employed in this thesis as a use case [139] for which the response time perceived by the end-users is important. To this end, a polynomial finite element method which has some computations is requested by the load generator. In our use case, a single task request normally takes 180 ms with our used experimental setup in conditions where the system is not overloaded. According to the explanation of the Alarm-Trigger component presented in Section 4.7 about how to determine the threshold for the service response time (T_{res}), we performed a set of experiments according to three different value: 180 ms, 190 ms and 250 ms. To this end, a slowly rising workload scenario for which the value defined for T_{res} is very significant has been investigated. During the experiment, the number of arrival requests smoothly rises from 100 to 600 task requests. Fig. 5.4 shows the number of new container instances allocated by three different auto-scaling methods using 180 ms, 190 ms and 250 ms for T_{res}.

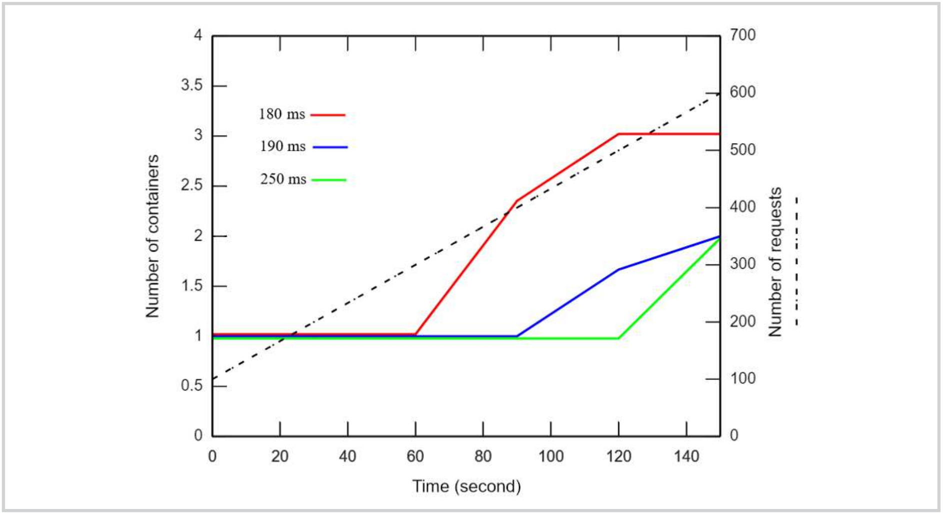


Figure 5.4

Number of containers allocated by three different auto-scaling methods using 180 ms, 190 ms and 250 ms for T_{res}.

Fig. 5.5 presents the response time offered by three different auto-scaling methods using 180 ms, 190 ms and 250 ms for T_{res}.

The response time provided by auto-scaling methods using 180 ms and 190 ms for T_{res} was nearly steady and identical. The average response time achieved by these two approaches was 184.5 ms and 184.88 ms, respectively. However, the auto-scaling method using 180 ms allocated more container instances on average (1.89) in comparison with

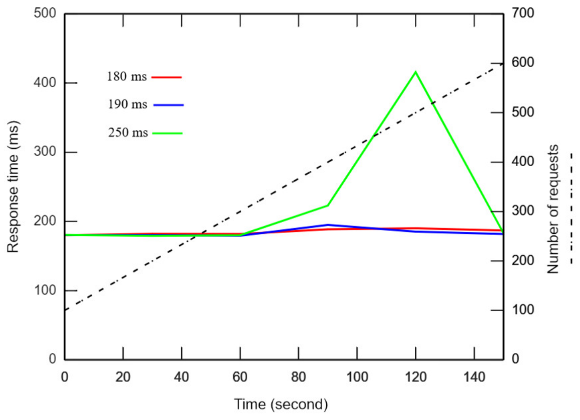


Figure 5.5

Response time provided by three different auto-scaling methods using 180 ms, 190 ms and 250 ms for T_{res}.

the approach using 180 ms (1.28) during the experiment. This is because the auto-scaling method using 180 ms (value next to the ordinary duration taken to accomplish a single task) was the first approach which reacted to the increasing workload. This method takes into account only infrastructure-level metrics without considering the service response. Therefore, the decision about adaptation actions depends mostly on how infrastructure-level metrics are changing over time—in which case the resources may be wasted, and hence the utilisation of allocated resources would be low.

In contrast, the auto-scaling method using 250 ms (much greater value than the ordinary time taken to perform one request) provided the slowest response time on average (250.52 ms). This is because this auto-scaling approach was the last method among all examined approaches reacted to the increasing workload. This method considers only the service response without paying attention to infrastructure-level metrics. As a consequence, the auto-scaling method using 250 ms for T_{res} allocated inadequate number of containers (1.17) during the experiment, and hence the application suffered from low performance.

Therefore, for the DM method in this thesis, in order to prevent application performance degradation, the service response time threshold (T_{res}) has been set to 190 ms that is neither very close to the value of usual time to execute a single job (180 ms) nor much bigger than this value. Consequently, the proposed DM auto-scaling method will be responsive to variations in not only infrastructure utilisation, but also service response time because T_{res} is not much bigger than the usual time to accomplish a single task request.

Here we discuss how the threshold (T_{res}) for this metric should be set in general. In order to make the system avoid any performance drop, the value of T_{res} should be set more than the usual time to process a single job without any issue when the system is not overloaded. In the case that T_{res} is set very close to the value of the usual time to process a single job, the auto-scaling method may lead to unnecessary changes in the number of running container instances, whereas the system is currently able to provide users an appropriate performance without any threat. Moreover, if T_{res} is set too much bigger than the value of the usual time to process a single job, the auto-scaling method will be less sensitive to application performance and more dependent on infrastructure utilisation.

It should be noted that if we deal with an execution environment in which the usual time to process a single job when the system is not overloaded is large, and the workload trend is even and predictable, the auto-scaling method has more chance to timely react to runtime changes in the workload density before a performance degradation occurs. In contrast, if the usual service time is too small, and the workload trend is occasional and unpredictable, the auto-scaling method may fail to provide the expected service quality for a while till the application performance will be improved.

One of main contributions of this thesis is proposing a new auto-scaling method which uses not only infrastructure, but also application-level monitoring data. Therefore, the cloud-based service provider needs to develop a Monitoring Agent able to measure the service time at runtime. If this duration taken to provide the service is not known, the auto-scaling method will be exclusively based on only infrastructure-level metrics. Such a case is similar to the auto-scaling method which uses a threshold for the service response time (T_{res}) close to the usual duration taken to execute a single request that may lead to an over-provisioning problem which wastes costly resources.

Conservative constant to avoid an unstable situation (α): In the downscaling principle defined in the Self-Adapter, α is a constant which helps the auto-scaling method conservatively prevent too frequent changes in the number of running containers due to minor fluctuations in the workload density. According to the explanation of the Self-Adapter component presented in Subsection 4.8 about how to define the value for the conservative constant (α) in order to avoid an unstable operational environment, we performed a set of experiments according to three different alpha: $\alpha = 0$, $\alpha = 5$ and $\alpha = 10$. To this end, a workload scenario has been examined that includes trembling number of requests at the beginning. Fig. 5.6 shows the number of container instances allocated at runtime by three different auto-scaling methods using $\alpha = 0$, $\alpha = 5$ and $\alpha = 10$. Moreover, Fig. 5.7 depicts the response time offered by three different auto-scaling methods using $\alpha = 0$, $\alpha = 5$ and $\alpha = 10$.

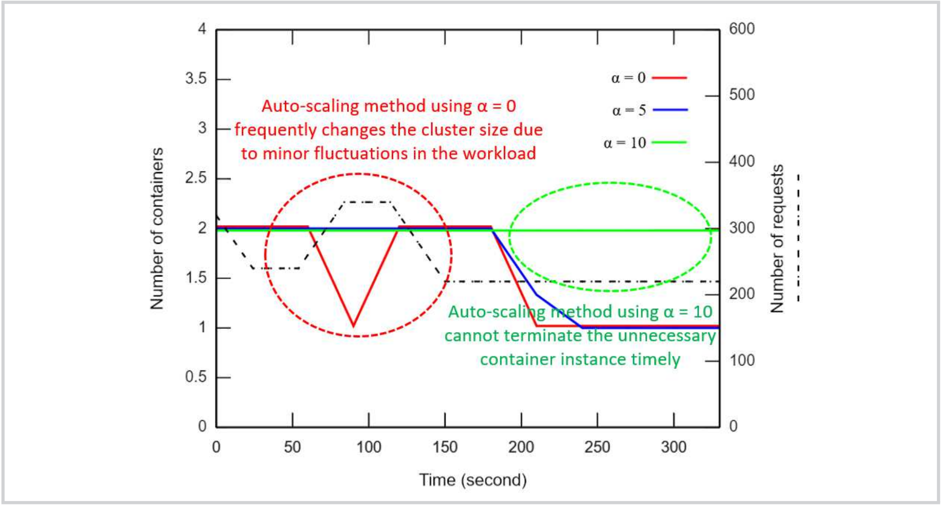


Figure 5.6

Number of containers allocated by three different auto-scaling methods using $\alpha = 0$, $\alpha = 5$ and $\alpha = 10$.

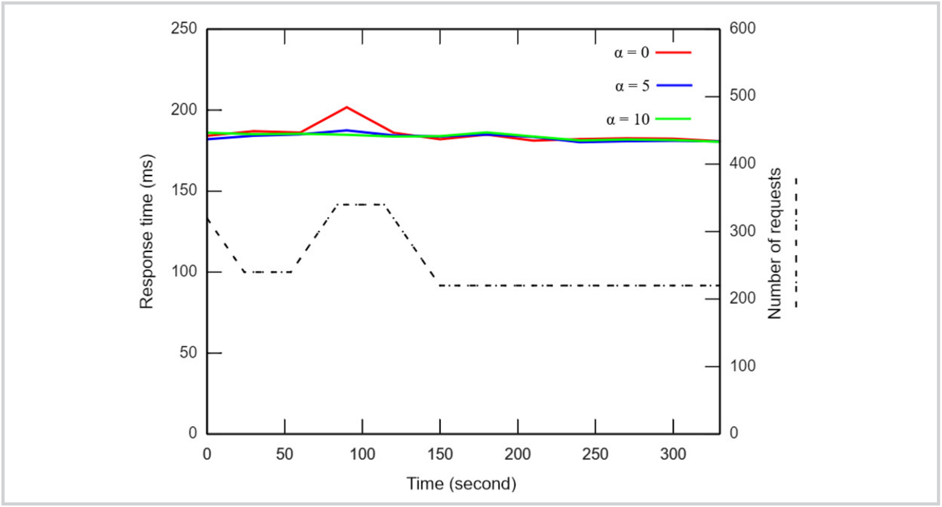


Figure 5.7

Response time provided by three different auto-scaling methods using $\alpha = 0$, $\alpha = 5$ and $\alpha = 10$.

Fig. 5.6 showed that the value of $\alpha = 0$ failed to provide the expected robustness of auto-scaling method. Since due to minor fluctuations in the workload, this auto-scaling approach stopped a container instance, and afterwards shortly started a new one again.

This fact, for a while, negatively affected the response time offered by the auto-scaling methods using α = 0, shown in Fig. 5.7. A value of α = 10 decreased the efficiency of the auto-scaling method because, in this case, the unnecessary container instance was not eliminated from the cluster at the right time. Consequently, a higher value of α would result in longer periods of over-provisioned resources. Therefore, the constant α used by the Self-Adapter component is set to the value of 5 which is able to prevent not only too frequent changes in the number of running containers due to minor fluctuations in the workload density, but also too much over-provisioning of resources.

Because the workload scenarios examined in our experiments are considered neither predictable nor even, the thresholds T_{cpu} and T_{mem} are set to 80%. Therefore, the DM auto-scaling method has enough chance to react to runtime fluctuations in the workload since these thresholds are not very close to 100%. Moreover, this fact prevents an over-provisioning issue because these thresholds are not less than 80%.

Tab. 5.1 shows the summary of parameters and their values mentioned above used in our experiments.

Tab. 5.2 shows the characteristics of all machines applied in our experiments. All these machines belong to the Academic and Research Network of Slovenia (ARNES) which is a non-profit cloud infrastructure provider. In our experiments, all host machines allocated to the cluster which provides the finite element analysis application have the same hardware characteristics. In addition to the machines assigned to the Load-Balancer and Monitoring Server, twelve hosts have been used in the cluster during the experiments.

Table 5.1

Parameters and their values applied in our experiments

Parameters	Description	Value
T_{cpu}	Threshold for the average CPU usage of the cluster	80%
T_{mem}	Threshold for the average memory usage of the cluster	80%
T_{res}	Threshold for the average response time	190 ms
α	Conservative constant to avoid an unstable situation	5
Monitoring interval	Time period between two observations of metrics	30 seconds
Adaptation interval	Time period between two possible successive adaptation events (increasing or decreasing the number of container instances in a cluster)	30 seconds

Table 5.2

Characteristics of infrastructures applied in our experiments

Feature	Load-Balancer	Monitoring Server	Hosts in the cluster
OS	Ubuntu 14.04	Ubuntu 14.04	Ubuntu 14.04
CPU(s)	4	2	4
CPU MHz	2397	2397	3100
Memory	16384 MB	4096 MB	4096 MB
Speed	1000 Mbps	1000 Mbps	1000 Mbps

5.3 *Significant performance properties*

When it comes to service response time guarantees, distinguishing the difference between auto-scaling methods in capability of offering response time under various workload scenarios is considered informative. In this regard, as shown in Tab. 5.3, the proposed DM auto-scaling method was compared with all other seven approaches using paired Student's t-tests with respect to all response time values over the experimental period for each workload scenario (n=145).

Table 5.3

P-values obtained by comparison of the DM method with other seven auto-scaling methods using paired t-tests with respect to all response time values over the experimental period for each workload pattern

Workload scenario	HPA	THRES1	THRES2	SS1	SS2	TTS1	TTS2
Slowly rising/falling	0.18800	0.14650	0.75633	0.00568	0.00033	0.00118	0.00009
Drastically changing	0.00055	0.00000	0.00000	0.00385	0.00000	0.00000	0.00000
On-off	0.00000	0.00191	0.00115	0.00000	0.00000	0.00000	0.00000
Gently shaking	0.00032	0.15528	0.00004	0.00051	0.63366	0.00000	0.00000
Real-world	0.00014	0.00718	0.00001	0.00000	0.00005	0.00424	0.00000

The 95th percentile value of response time, listed in Tab. 5.4, is an appropriate indicator of the auto-scaling methods' capability for delivering Quality of Service according to a Service Level Agreement (SLA). Moreover, the median response time obtained by all auto-scaling methods in every workload scenario is presented in Tab. 5.5.

Tab. 5.6, Tab. 5.7, Tab. 5.8, Tab. 5.9 and Tab. 5.10 show the resource utilisation achieved by all examined auto-scaling methods for each workload scenario with respect to average number of container instances as well as average CPU and memory usage. In these tables,

Table 5.4

The 95th percentile of the response time achieved by all investigated auto-scaling methods in every workload pattern

Workload scenario	HPA	THRES1	THRES2	SS1	SS2	TTS1	TTS2	DM
Slowly rising/falling	213.07	202.40	208.21	364.70	372.90	365.20	398.90	207.40
Drastically changing	652.82	659.90	619.20	852.06	1623.14	1609.22	1270.98	410.28
On-off	471.75	386.00	387.90	683.70	493.60	550.40	566.50	232.60
Gently shaking	201.83	196.04	195.89	194.80	195.00	268.69	240.47	194.85
Real-world	204.64	208.84	214.26	202.94	233.64	215.66	260.2	197.32

Table 5.5

The median response time achieved by all investigated auto-scaling methods in every workload pattern

Workload scenario	HPA	THRES1	THRES2	SS1	SS2	TTS1	TTS2	DM
Slowly rising/falling	190.6	189.6	188.7	190.7	192.9	189.6	192.1	191.2
Drastically changing	185.6	193.0	199.1	190.5	197.9	199.4	201.0	190.1
On-off	199.6	191.3	195.5	194.5	200.7	209.7	211.4	190.5
Gently shaking	189.9	186.1	188.7	187.9	184.9	196.1	196.7	185.3
Real-world	193.4	192.0	194.4	195.3	193.9	195.3	197.4	192.3

the column named resource utilisation function equals to the average number of containers multiplied by the 95th percentile of the response time provided by auto-scaling approaches for each workload scenario. In order to appropriately enhance the resource utilisation of an auto-scaling method while providing acceptable service response time, the value of this function needs to be reduced as much as feasible.

Tab. 5.6, which is explained in Section 5.4, shows the average number of container instances as well as average CPU and memory usage of the new DM method compared with seven other auto-scaling approaches for the slowly rising/falling workload pattern.

Tab. 5.7, which is explained in Section 5.5, shows the average number of container instances as well as average CPU and memory usage of the new DM method compared with seven other auto-scaling approaches for the drastically changing workload pattern.

Table 5.6

Comparing the new DM method with existing auto-scaling methods with respect to resource utilisation for the slowly rising/falling workload pattern

Method	Average number of containers	Average CPU usage	Average memory usage	Resource utilisation function
DM	3.47	64.36	31.55	719.68
HPA	3.25	65.48	31.60	692.48
THRES1	3.65	62.24	31.50	738.76
THRES2	3.52	64.84	31.64	732.90
SS1	4.36	55.85	31.57	1590.09
SS2	3.84	61.86	31.73	1431.94
TTS1	3.12	71.85	31.56	1139.42
TTS2	3.32	70.78	31.58	1324.35

Table 5.7

Comparing the new DM method with existing auto-scaling methods with respect to resource utilisation for the drastically changing workload pattern

Method	Average number of containers	Average CPU usage	Average memory usage	Resource utilisation function
DM	3.71	41.07	31.69	1522.14
HPA	2.50	50.77	31.68	1632.05
THRES1	3.31	40.72	31.58	2184.27
THRES2	3.31	40.92	31.51	2049.55
SS1	3.53	41.54	31.43	3007.77
SS2	2.47	45.22	31.68	4009.15
TTS1	2.68	45.69	31.63	4312.71
TTS2	2.68	45.84	31.71	3406.23

Tab. 5.8, which is explained in Section 5.6, shows the average number of container instances as well as average CPU and memory usage of the new DM method compared with seven other auto-scaling approaches for the on-off workload pattern.

Table 5.8

Comparing the new DM method with existing auto-scaling methods with respect to resource utilisation for the on-off workload pattern

Method	Average number of containers	Average CPU usage	Average memory usage	Resource utilisation function
DM	3.58	53.49	31.40	832.71
HPA	2.77	66.90	31.50	1306.75
THRES1	3.39	57.25	31.55	1308.54
THRES2	3.40	58.27	31.74	1318.86
SS1	3.23	51.50	31.61	2208.35
SS2	2.71	58.53	31.72	1337.66
TTS1	2.33	64.90	31.69	1282.43
TTS2	2.33	64.65	31.70	1319.94

Tab. 5.9, which is explained in Section 5.7, shows the average number of container instances as well as average CPU and memory usage of the new DM method compared with seven other auto-scaling approaches for the gently shaking workload pattern.

Table 5.9

Comparing the new DM method with existing auto-scaling methods with respect to resource utilisation for the gently shaking workload pattern

Method	Average number of containers	Average CPU usage	Average memory usage	Resource utilisation function
DM	4.00	66.67	31.75	779.40
HPA	3.78	70.46	31.80	762.92
THRES1	4.00	66.94	31.58	784.16
THRES2	3.68	72.10	31.61	720.87
SS1	4.25	64.31	31.49	827.90
SS2	4.00	67.74	31.76	780.00
TTS1	3.41	79.25	31.62	916.23
TTS2	3.38	78.95	31.63	812.79

Tab. 5.10, which is explained in Section 5.8, shows the average number of container instances as well as average CPU and memory usage of the new DM method compared with seven other auto-scaling approaches for the real-world workload pattern.

Table 5.10

Comparing the new DM method with existing auto-scaling methods with respect to resource utilisation for the real-world workload pattern

Method	Average number of containers	Average CPU usage	Average memory usage	Resource utilisation function
DM	10.15	72.38	31.59	2002.80
HPA	9.20	75.81	31.55	1882.69
THRES1	9.86	71.02	31.61	2059.16
THRES2	9.98	70.32	31.56	2138.31
SS1	10.16	73.15	31.60	2061.87
SS2	9.38	74.23	31.59	2191.54
TTS1	7.27	79.09	31.60	1567.85
TTS2	7.64	75.44	31.62	1987.92

5.4 *Slowly rising/falling workload scenario*

In this pattern as depicted in Fig. 5.8, the workload consists of two steps. During the first step of the workload pattern, the number of arrival requests smoothly rises from 100 to 1500 task requests per six seconds. Later on, in the second step, workload density drops down slowly from 1500 to 100 requests. Fig. 5.8 demonstrates that the number of container instances increases during the first step of the workload pattern, and it decreases during the second step with respect to the number of incoming requests over time by all eight auto-scaling methods.

For the slowly rising/falling workload pattern, the paired t-tests comparing DM with HPA, THRES1 and THRES2 show no statistically significant difference with p ¿ 0.01. While all provisioning methods are capable of providing acceptable performance on average, the service response time provided by TTS1 and TTS2, SS1 and SS2, is occasionally low in comparison with DM, THRES1, THRES2 and HPA. Reason for this is that the adaptation interval adopted in TTS1, TTS2, SS1 and SS2 is one minute versus 30 seconds employed in DM, THRES1, THRES2 and HPA. Therefore, the service response time may be unsuitable for a while in some situations if the adaptation interval is not short enough, as depicted in Fig. 5.9. This fact caused relatively weaker application performance provided by TTS1, TTS2, SS1 and SS2 compared with DM, THRES1, THRES2 and HPA in terms of the 95th percentile values.

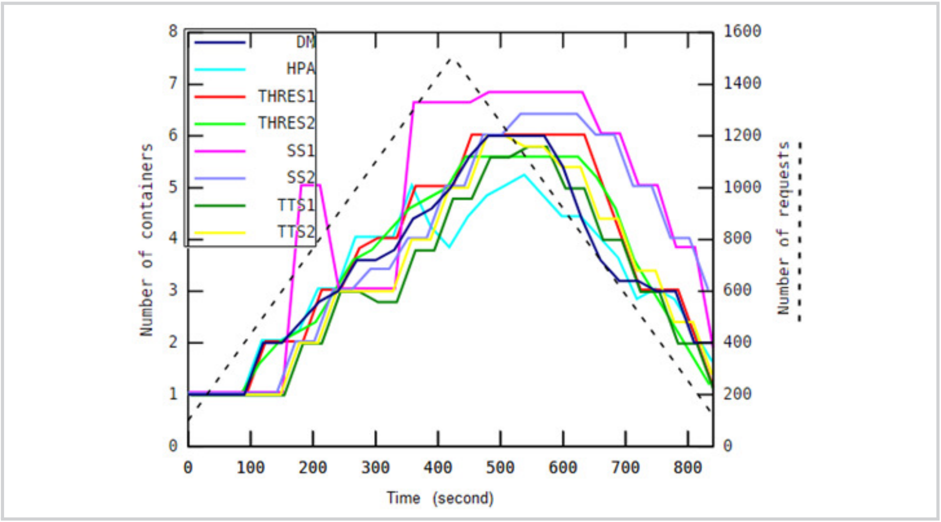


Figure 5.8

Dynamically changing number of container instances in response to a slowly rising/falling workload pattern.

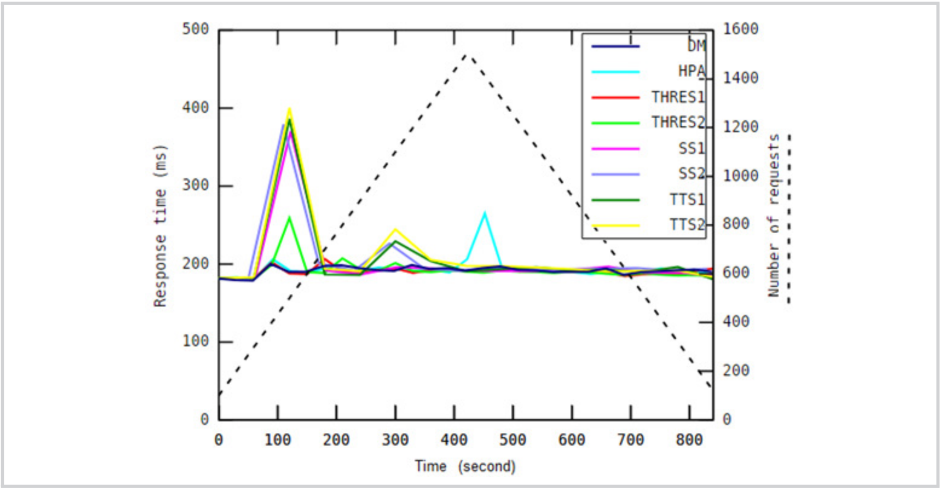


Figure 5.9

Average response time of the application in response to a slowly rising/falling workload pattern.

The length of the adaptation interval, whether one minute or 30 seconds, adopted by auto-scaling approaches influences the overall performance of the application. For instance when time was 90 s and before the CPU run queue started filling up (96%), the DM method decided to launch one new container instance due to the increase in the workload density. Thus, the service response time provided by the DM method was not inappropriately influenced by the increase in the workload. If we consider the SS1 auto-scaling method, in the situation when time was 120 s and after the system was overloaded as a result of the growing workload, SS1 added four new container instances to the cluster. Although, at this time the CPU utilisation was already at almost 100%, and thus the slow response time was offered by SS1 for a while. Currently, the service cluster contains five containers. This cluster size (five container instances) is more than what is required to process the current workload. Consequently, this decision was reverted after a while when time was 240 s, and two container instances were stopped.

In this workload scenario, DM, THRES1, THRES2 and HPA allocated almost the same number of containers and hade nearly the same level of average resource utilisation in terms of CPU and memory usage. The SS1 adaptation approach employed more container instances (4.36) compared to all other auto-scaling methods.

Furthermore, it can be simply concluded that the finite element analysis application is not a memory-intensive service, as the average memory utilisation was nearly steady during the whole conducted experiments, and the same for all auto-scaling methods—almost 31% of the entire memory.

5.5 Drastically changing workload scenario

In this scenario, a drastic increase appears in the workload intensity. During this experiment, depicted in Fig. 5.10, the number of incoming requests rises suddenly from 100 to 1500, and after a while it instantly drops down back to 100 task requests again. For this workload scenario, the paired t-tests indicated that there is a statistically significant difference between the DM method and all other auto-scaling approaches. Fig. 5.10 illustrates that the proposed DM method suitably recognised the sudden fluctuation in the workload density and then attempted to timely launch enough new containers at the beginning of sudden workload surge faster than other provisioning mechanisms. Hence, for this examined workload scenario, our proposed DM method is the only mechanism capable of providing relatively acceptable application QoS in terms of the 95th percentile of the response time distribution.

Afterwards, when the workload density suddenly drops down again to 100 task requests per six seconds, all auto-scaling methods, except SS1 and HPA, do not stop unnecessary containers running in the service cluster at once, and hence the number of container instances slightly declines in consecutive intervals. The DM method was the auto-scaling mechanism which provisioned more containers than other adaptation approaches. During this experiment, the average number of container instances allocated by the DM method was 3.71.

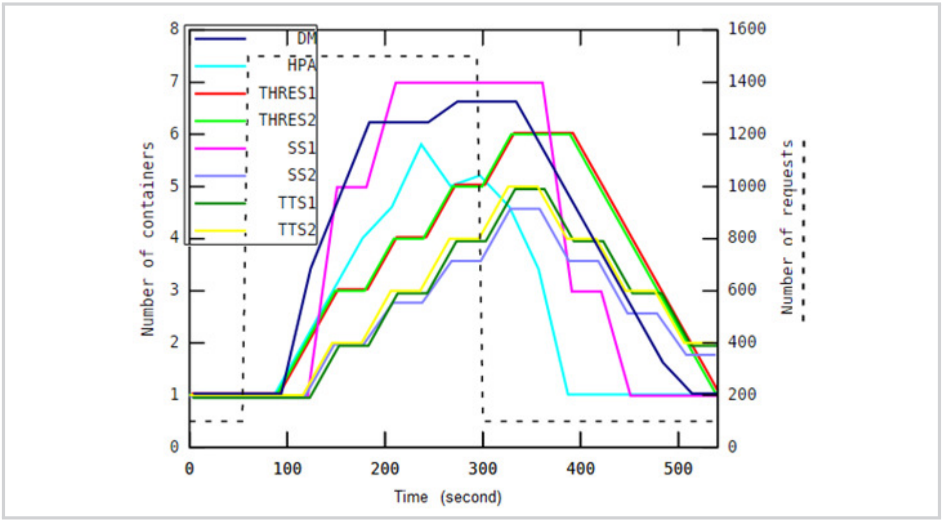


Figure 5.10

Dynamically changing number of container instances in response to a drastically changing workload pattern.

Fig. 5.11 displays that the service response time offered by the proposed DM method, compared to other approaches, is less inappropriately affected by the drastic change in the workload.

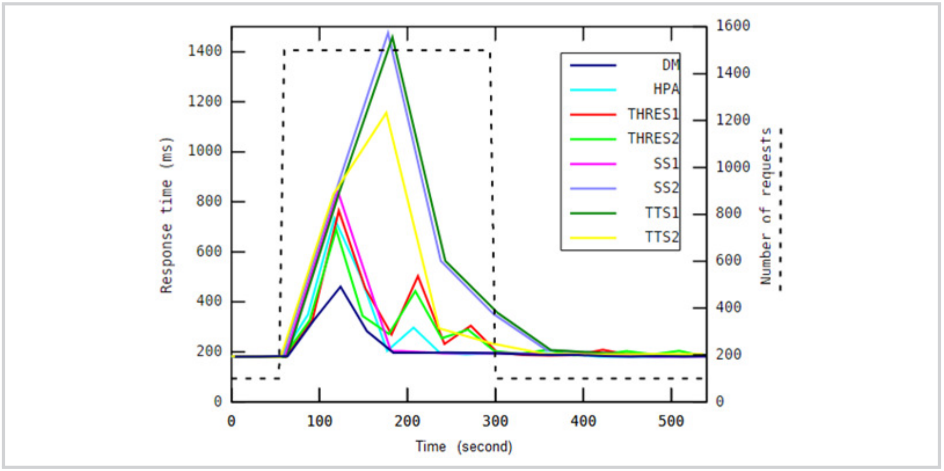


Figure 5.11

Average response time of the application in response to a drastically changing workload pattern.

Once more, the amount of average memory utilisation was almost steady (31% of the whole memory capacity), and the same for all auto-scaling methods during the entire conducted experiment in this workload pattern, considered as additional confirmation of a slowly rising/falling workload pattern's result, indicating that the conducted service (finite element analysis application) is not a memory-intensive benchmark.

5.6 On-off workload scenario

During this conducted experiment, the on-off workload scenario includes three active periods. The active periods consist of, respectively, 1500, 1200, and 700 task requests per six seconds (shown in Fig. 5.12). Inactive periods between peak spikes are 30 seconds. In the on-off workload scenario, the paired t-tests implied a statistically significant difference in the means of response time metric provided by all auto-scaling methods. The only method capable of timely provisioning an adequate number of containers to address peak spikes is DM. This is because the proposed DM method is more agile than other auto-scaling mechanisms in order to instantiate required containers at the commencement of sudden workload surges, and also this method does not terminate most of the container instances immediately when each peak spike disappears. Therefore, the DM method allocated more containers on average (3.58) than other auto-scaling mechanisms in the on-off workload scenario.

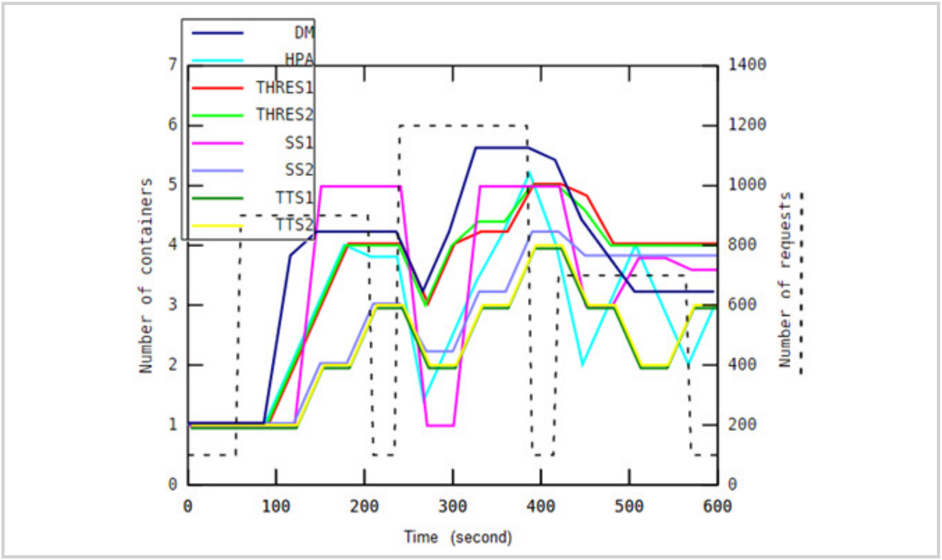


Figure 5.12

Dynamically changing number of container instances in response to an on-off workload pattern.

The benefit of using 30-second adaptation interval instead of 1-minute adaptation interval can be recognised in Fig. 5.12. At the commencement of the first active period, DM and SS1 made similar decision to launch new container instances because of the sudden increase in the workload density. The DM method allocated three extra container instances starting from time=90 s whereas SS1 assigned four new container instances when the system is already overloaded at time=120 s, that means 30 seconds later than time=90 s. In such situation, the core competency of DM compared with SS1 lies in its agility to timely adapt the service performance to the unexpected increase in the workload density. Therefore in this workload pattern, the difference between DM and SS1 in terms of service response time can be taken into consideration enormous. That is why the worst response times offered by DM and SS1 during the first active period were 225.04 ms versus 558.88 ms, respectively.

In the on-off workload scenario, the median and the 95th percentile of the response time achieved by DM in this examined experiment were 191.2 ms and 232.60 ms, respectively, that can be considered acceptable with respect to end-users' satisfaction. Whereas sudden active periods inappropriately result in an increase in the service time of the task requests for the other seven auto-scaling approaches, as depicted in Fig. 5.13. Compared to DM, the 95th percentile values of the response time provided by all other provisioning approaches are very slow that can be considered unsuitable.

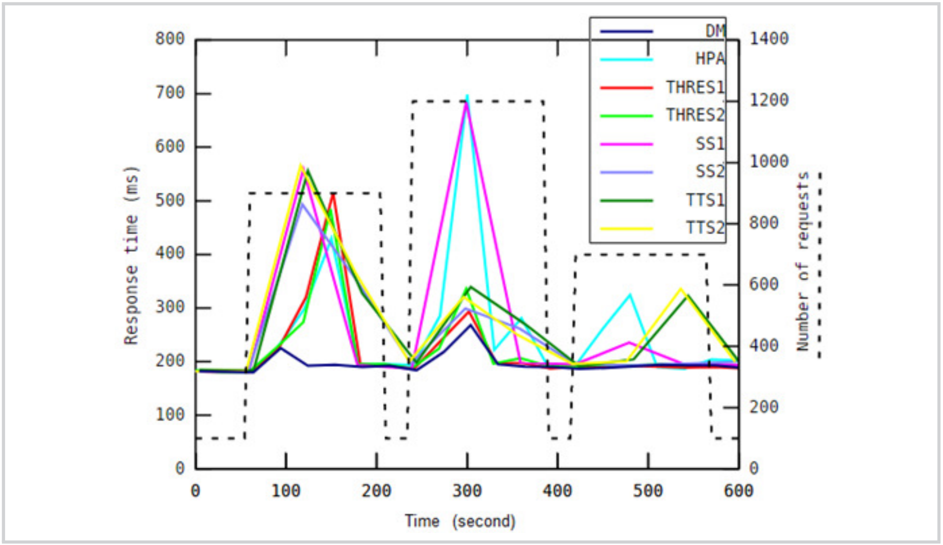


Figure 5.13

Average response time of the application in response to an on-off workload pattern.

During this conducted experiment, the average memory utilisation was found to be constant (31%) for all auto-scaling mechanisms, and it did not change significantly while having sudden fluctuations in the number of requests over time.

5.7 *Gently shaking workload scenario*

In this experiment, the pattern is a trembling workload which does not vary drastically. As depicted in Fig. 5.14, it frequently changes between 700 and 1000 task requests to be accomplished by the application. Fig. 5.14 implies that if the workload density does not vary drastically, there would be neither increment nor decrement in the number of running container instances for DM, SS2 and THRES1. This is why, for this workload scenario, the paired t-tests comparing DM with SS2 and THRES1 indicated that we cannot reject the zero hypothesis ($p>0.01$). Or in other words, the DM method behaves the same way as the SS2 and THRES1 methods. Moreover, the number of container instances did not change to a great extent by other auto-scaling methods namely SS1, THRES2, HPA, TTS1, and TTS2.

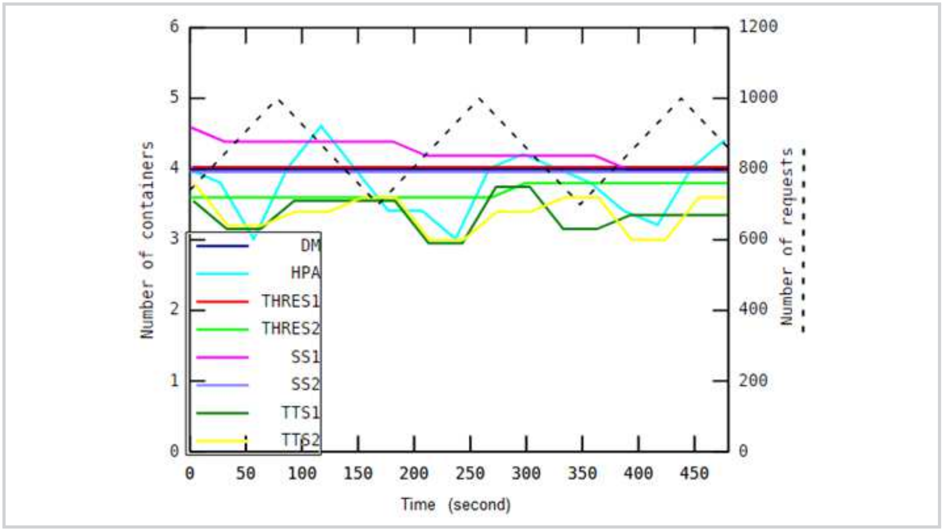


Figure 5.14
Dynamically changing number of container instances in response to a gently shaking workload pattern.

The SS1 approach allocated more containers on average (4.25) than other auto-scaling methods, whereas the average service response time offered by all adaptation mechanisms was almost consistent and identical for this workload pattern, shown in Fig. 5.15.

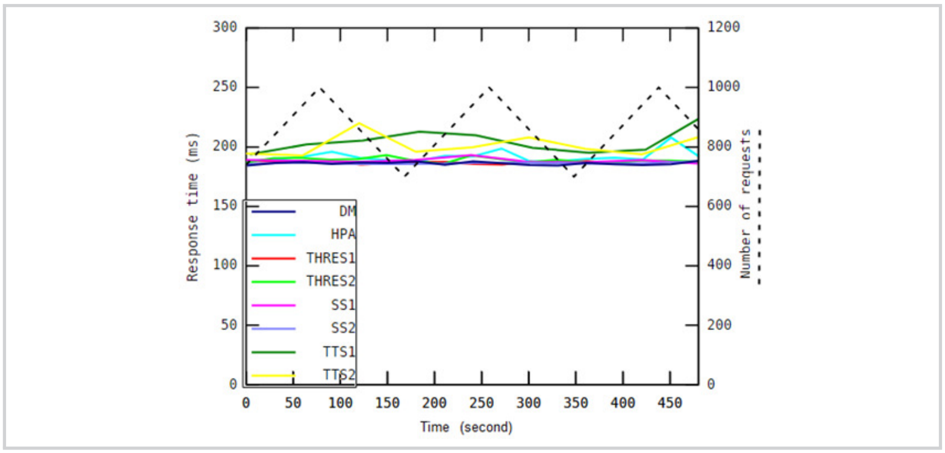


Figure 5.15

Average response time of the application in response to a gently shaking workload pattern.

As a result, allocating more container instances by the SS1 auto-scaling method in this workload pattern inappropriately caused resource under-utilisation, in terms of less average CPU resource utilisation. The average CPU usage was reported 64.31% in comparison with what was obtained by other auto-scaling approaches. Although, all auto-scaling methods has almost the same level of memory usage (31%) in this experiment.

5.8 Real-world workload scenario

Besides the previous workload scenarios, in order to validate the applicability of our proposed DM auto-scaling method against real-world situations, FIFA World Cup 98 workload dataset [140] was applied in this thesis. This workload trace has been widely adopted in various auto-scaling research works [87, 141–145] so far. For our experiment, a 20-minute trace (shown in Fig. 5.16) on the 12th of July 1998 starting at 20:30:00 was used. The number of arrival requests is varied between 2112 and 2858 during this time period that demonstrates a large variance (750) in the workload over time.

In order to adapt the application QoS to the varying workload density and provide a favourable performance, the number of running containers provisioned by auto-scaling approaches changes at runtime. DM and SS1 allocated the same amount of resources in terms of containers on average for the real-world workload scenario. For both DM and SS1, the average number of container instances was equal to 10.1. Other approaches assigned fewer containers compared with the DM and SS1 methods in this experiment.

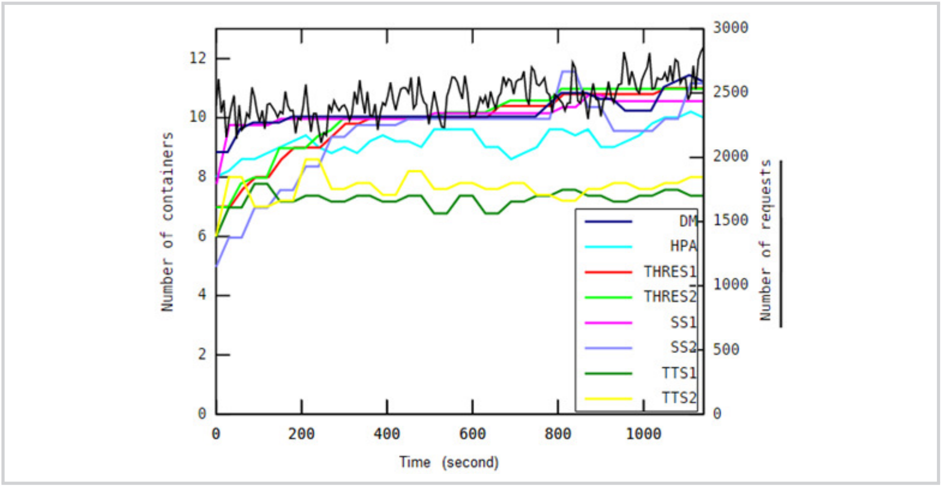


Figure 5.16

Dynamically changing number of container instances in response to a real-world workload pattern.

Fig. 5.17 presents the average service response time offered by all eight auto-scaling methods during this real-world workload scenario.

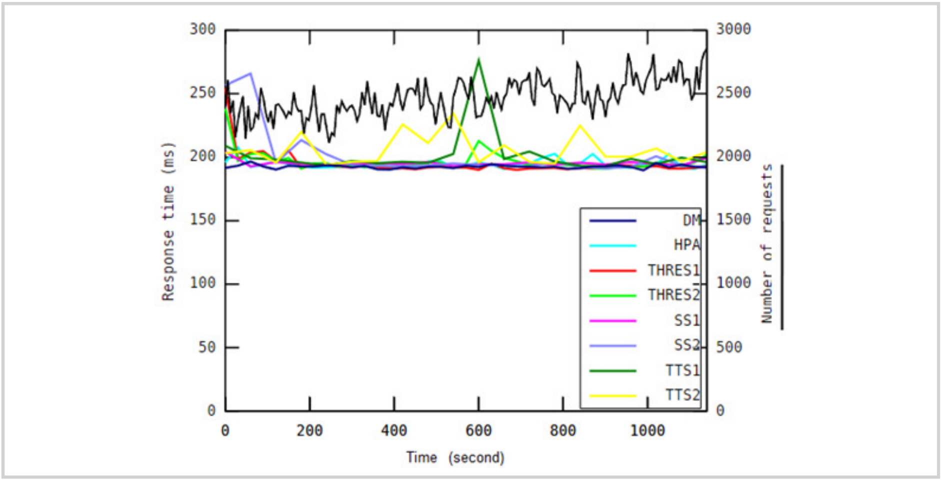


Figure 5.17

Average response time of the application in response to a real-world workload pattern.

In the real-world workload scenario, the service response time provided by the proposed DM method was not inappropriately influenced by the workload fluctuations, since it was thoroughly constant in comparison with what was offered by other mechanisms. It means that there is no big difference between the 95th percentile of the response time distribution (197.32 ms) and the median response time (192.3 ms) achieved by the DM method.

Analogous to the result obtained in previous workload scenarios, the experiment in this workload pattern also re-indicates that the memory resource usage of the cluster does not have any effect on the finite element analysis application's performance in spite of the changing number of arrival task requests during execution. This is because the average memory usage was 31% for all auto-scaling mechanisms at runtime. This fact fortunately allows the cloud-based application provider to accomplish efficient memory allocation allocated for running containers in advance.

5.9 *Summary of results*

As already explained, the resource utilisation function is defined as the average number of containers multiplied by the 95th percentile of the response time. For every auto-scaling mechanisms, all values of this function obtained in each workload scenario were summed together to form an overall score. The calculated scores are: DM = 5856.73, HPA = 6276.89, THRES1 = 7074.89, THRES2 = 6960.49, SS1 = 9695.98, SS2 = 9750.29, TTS1 = 9218.65 and TTS2 = 8851.23. These results imply that the proposed DM method is the best among eight examined mechanisms. This is because the DM method provided the minimum overall score in comparison with the other auto-scaling mechanisms. In other words, the proposed DM method is capable of avoiding over-provisioning of resources while being able to offer optimal application performance in terms of the service response time. Considering all workload patterns investigated in this thesis, the competence of DM is its ability to use a multi-level monitoring platform and timely adjust the auto-scaling rules to changes in the workload at runtime.

The Cumulative Distribution Function (CDF) of service response time achieved by all examined auto-scaling methods is shown in Fig. 5.18, Fig. 5.19, Fig. 5.20, Fig. 5.21 and Fig. 5.22 for each workload scenario. According to all these figures, it can be concluded that the proposed DM method performs better than other auto-scaling approaches since it has higher probability to provide desired service response time under varied amount of workloads, and thus enhance the application performance. The probability that the response time offered by the DM method would be slow is almost zero for all workload patterns, except for the drastically changing scenario.

Fig. 5.19 indicates that the response time offered by the DM method was relatively more acceptable than other seven provisioning approaches in the drastically changing workload scenario. During this workload scenario, the probability of service response time being fast provided by other auto-scaling mechanisms is significantly small.

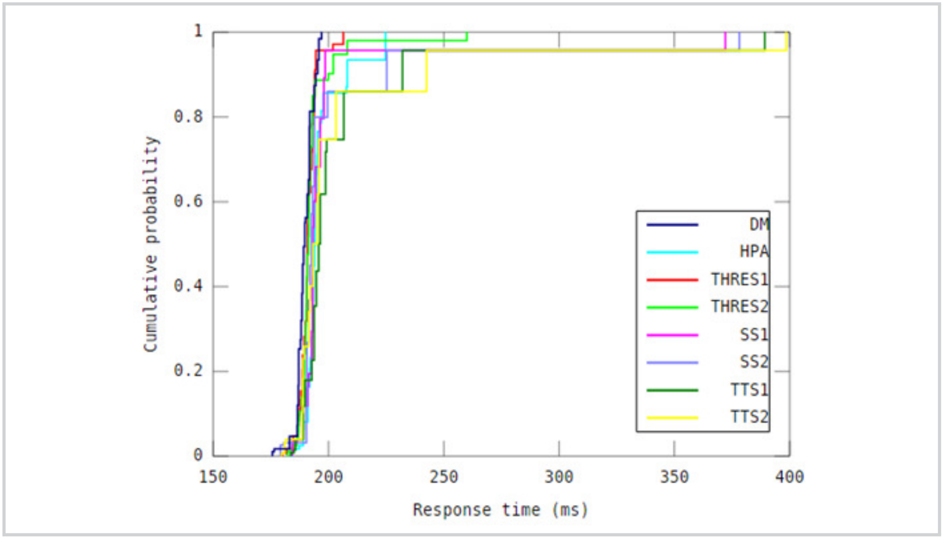


Figure 5.18

CDF of service response time observed in the slowly rising/falling workload scenario.

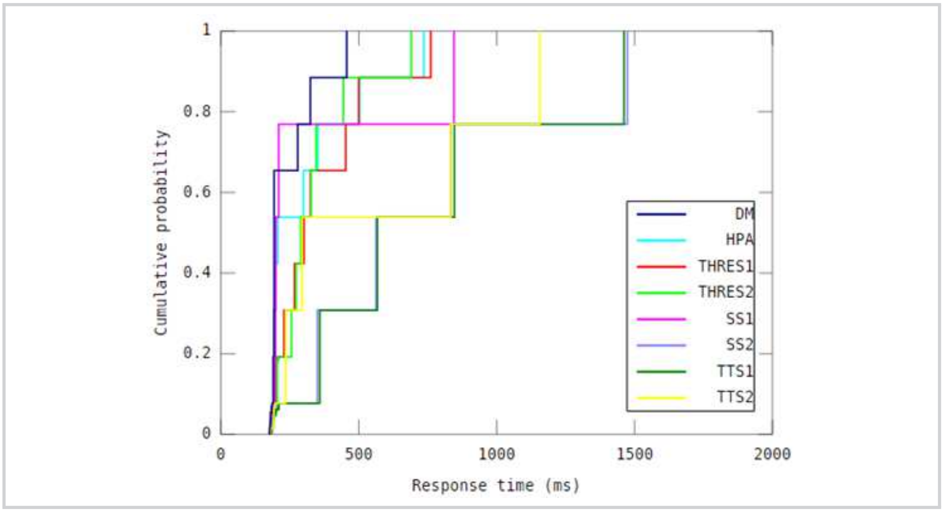


Figure 5.19

CDF of service response time observed in the drastically changing workload scenario.

Fig. 5.20 shows the CDF of service response time offered by all investigated auto-scaling approaches in the on-off workload scenario.

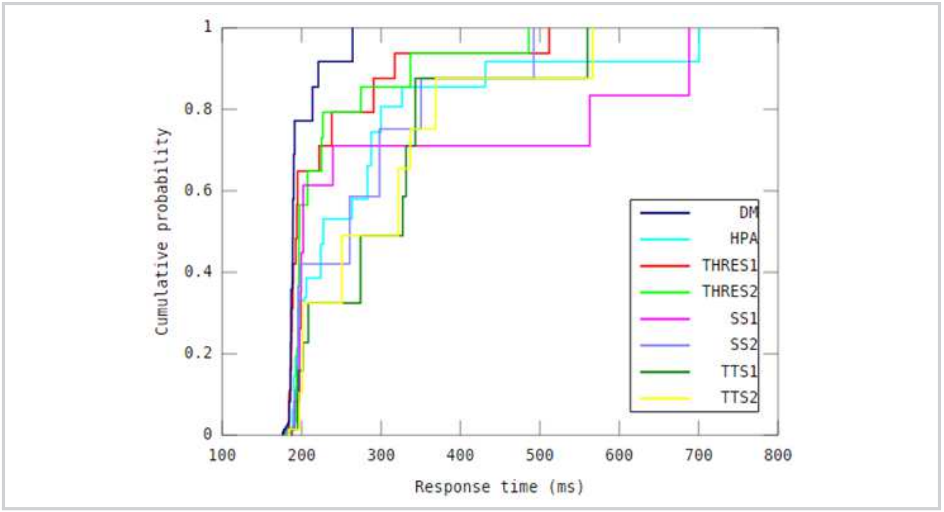


Figure 5.20

CDF of service response time observed in the on-off workload scenario.

Fig. 5.21 shows the CDF of service response time offered by all investigated auto-scaling approaches in the gently shaking workload scenario.

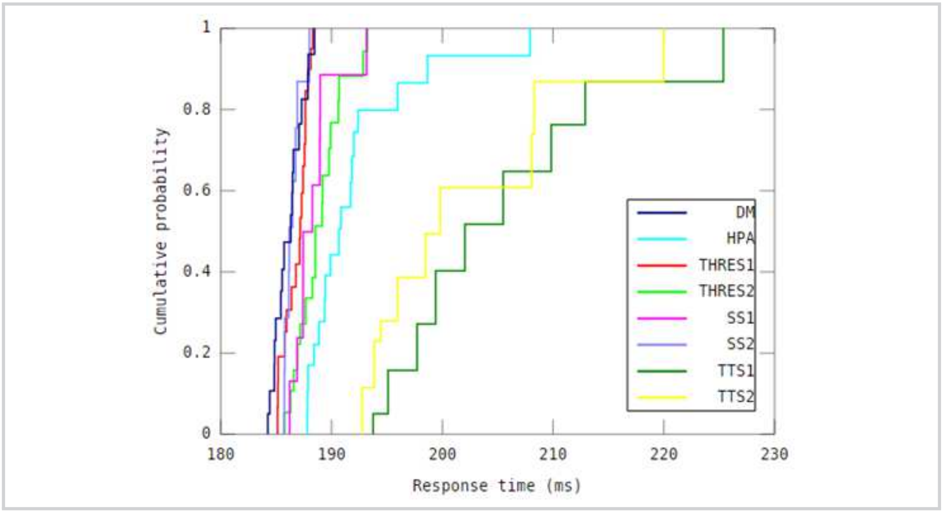


Figure 5.21

CDF of service response time observed in the gently shaking workload scenario.

Fig. 5.22 shows the CDF of service response time offered by all investigated auto-scaling approaches in the real-world workload scenario.

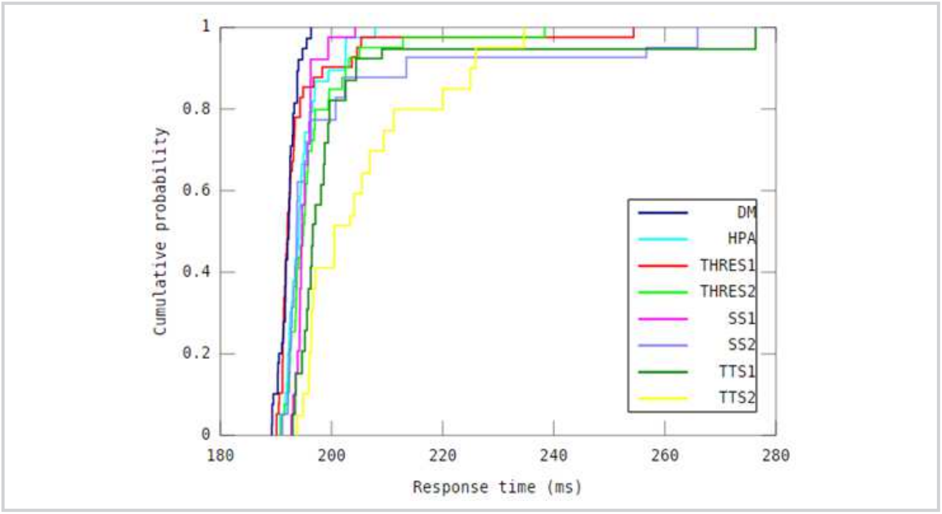


Figure 5.22

CDF of service response time observed in the real-world workload scenario.

A challenge in designing a monitoring framework in the cloud environment is ensuring that the overhead of the monitoring system is kept to the minimum. The distributed nature of proposed monitoring framework quenches the runtime overhead of system to a number of Monitoring Agents running across container instances. Host machines provisioned in order to run the containerised application in this thesis had hardware characteristics as 4-core CPU with 4 GB of RAM and 80 GB of storage. A detailed view on the resource consumption of the Monitoring Agent revealed that our approach is lightweight in terms of CPU and memory overhead. To confirm this, we applied the *top* tool which provides a dynamic real-time view of tasks currently being managed by the Linux kernel. Our running Monitoring Agent used in this work consumes only 0.3 percent of the whole CPU time and 3.1 percent of the whole memory usage on average.

Furthermore, the developed Monitoring Agent used in this work consumes a small fraction of network bandwidth. To this end, we parsed the output of *nethogs* tool to estimate the bandwidth overhead introduced by our Monitoring Agent. We found out our Monitoring Agent transmits 1282944 bytes during 15 minutes, which means $\sim$712 bytes per second on average.

6

Discussion

The previous chapter (Chapter 5) described the results obtained after empirical experiments performed as a set of proof-of-concept tests. The current chapter provides a discussion behind the results explained in Chapter 5. The archived results can be used for the analysis of the implemented auto-scaling DM method and its limitations, its advanced usability in the software engineering domain, comparison with other production rule-based methods, and the substantial level of improvement in the effectiveness of the auto-scaling process applied to handle different workload patterns.

As described before, there exist many auto-scaling methods being used by commercial cloud providers, although they mainly use static, single-level auto-scaling rules which cannot be flexible to adjust themselves to the status of the execution environment at runtime. Moreover, such existing auto-scaling methods are not able to employ both infrastructure and application-level policies for dynamically launching new container instances or terminating unrequired container instances.

In this thesis, we proposed a new Dynamic Multi-level (DM) auto-scaling method with dynamically changing thresholds. The proposed DM method uses not only infrastructure, but also application-level monitoring data in order to allocate the optimal amount of resources (with regard to the necessary number of container instances) needed to ensure application performance (with regard to the service response time) with neither resource under-provisioning nor over-provisioning.

In Chapter 5, our proposed DM method was compared with seven existing rule-based auto-scaling approaches in different synthetic and real-world workload scenarios. These approaches include HPA (Horizontal Pod Auto-scaling), THRES1 (1st method of THRESHOLD), THRES2 (2nd method of THRESHOLD), SS1 (1st method of Step Scaling), SS2 (2nd method of Step Scaling), TTS1 (1st method of Target Tracking Scaling), and TTS2 (2nd method of Target Tracking Scaling). In comparison to seven other approaches, the proposed DM method achieved the best overall score with regard to avoiding over-provisioning of resources while providing optimal application performance in terms of the response time. Regarding all workload scenarios examined in this work (slowly rising/falling workload pattern, drastically changing workload pattern, on-off workload pattern, gently shaking workload pattern, real-world workload pattern), the core competency of the DM method is its ability to use a multi-level monitoring framework and also dynamically adjust itself to changes in the workload density.

The monitoring interval is a significant part which has been scrutinised in this thesis. Choosing an effective monitoring interval is needed to make sure the reliability of the entire system, to prevent excessive overhead, and to avoid losing control over the execution environment during auto-scaling events. Setting up an applicable measurement interval is a challenging subject. This is because a low level of measurement ratio may result in missing dynamic changes in the running environment, and thus the system is not able to adapt itself to a new situation in order to carry on its operation without any application performance degradation.

In some operational scenarios, the difference between the average service response time and the monitoring interval may lead to stability problems to the elasticity management, which is not the case for many applications such as finite element analysis. As an example, for video streaming real-time services, any violation of application QoS constraints should be monitored cautiously, because any violation should not be overlooked. Hence, the monitoring interval needs to be short enough to sufficiently capture all required characteristics of the application at runtime. Furthermore, self-adaptation of such services also needs an acceptable level of agility, which has currently gained an increasing range of attention as a research area that still required to be totally improved.

Moreover, the developed DM auto-scaling method can be extended to allow for vertical scaling of container instances [146]. Vertical scaling function represents an adaptation operation to resize processing power, memory capacity, or bandwidth resource allocated to running containers depending on runtime variations in the workload density. Although in this manner, the maximum amount of such resources available for each container instance is restricted to the capacity of the host machine. Accordingly, the combination of horizontal and vertical scaling mechanisms may be used to the same service in order to benefit from both techniques. Although, some applications such as Java/J2EE solutions [147] are not capable of dynamically managing the memory allocation even if the memory capacity is resized at the Operating System (OS) or infrastructure level. In such solutions, the applications should be restarted with new resized capacity of memory when vertical scaling happens that is not appropriate within many real-world cases.

The conducted experiments in this thesis are based upon Docker technique, although the presented auto-scaling architecture may be implemented in other container-based virtualization technologies such as LXC [148], OpenVZ [149], and lmctfy [150]. The reason is that all functionalities specified in both Self-Adapter and Alarm-Trigger, and also the StatsD protocol used to transmit, collect and aggregate monitoring information related to the infrastructure or application, are independent from not only providers of underlying cloud infrastructure, but also containerisation technologies.

The developed multi-level monitoring platform implemented to be employed in this thesis is able to monitor various container-level parameters including CPU, memory, bandwidth, and disk [151]. This monitoring platform was used to measure disk and bandwidth for a container-based file upload use case in our previous research work [150].

The main objective of an auto-scaling method is to meet the desired response time by allocating the right amount of resources. Along this line, while monitoring infrastructure-level metrics such as CPU and memory, the proposed DM auto-scaling method considers the service response time to avoid both under-provisioning and over-provisioning problems. Hence, for completeness, we consider both CPU and memory in the proposed method. In this thesis, a finite element analysis application which is a CPU-intensive service was employed as a use case examined in auto-scaling experiments. The DM method can certainly specify scaling policies for memory resources that are similar to CPU resources. However, in practice, for memory-intensive applications such as in-memory databases (e.g. HSQLDB), such proposed auto-scaling may not be very useful for memory-intensive applications because of some key reasons:

- In order to scale a memory-intensive application, the data in memory should be shifted from one node to another one which is very time and bandwidth consuming. Therefore, unlike CPU, scaling memory is much more expensive.
- A lot of applications are not designed for dynamic memory size. For example, memory-intensive applications primarily determine the buffer cache size when starting, which is then fixed for the rest of time, and hence changing memory size does not make sense for them. We can certainly rely on swapping of the OS to make it transparent, but performance will not be appropriate nevertheless.
- In order to run memory-intensive applications, using high-memory machines in advance needs to be considered as a significant requirement to deliver fast performance for workloads that process large data sets in memory.

Over the entire course of experimentation, different threats to the validity of the results have been analysed as follows:

- Variations observed in running conditions (e.g. time-varying processing delays, CPU and I/O load factors, etc.) may slightly influence the archived results presented in Chapter 5. To obtain a greater validity of results, each conducted experiment on each workload scenario was repeated five times to prevent this threat. Consequently, the presented results are reported as average values over independent runs.

- QoS properties of cloud-based infrastructure such as availability, bandwidth quality, etc. may vary at runtime, independent from the workload characteristics. Thus, when a container instance should be launched and deployed on a host machine, the cloud-based application provider requires to ensure that the host machine is capable of fulfilling the needed requirements of the container-based time-critical application. In this regard the performance of running infrastructures also need be continuously characterised. This function is facilitated by our developed multi-level monitoring platform at present.

- Different additional external factors (e.g. the mobility of the clients, unstable network conditions at the side of end-users, and client' network channel diversity, and so forth) may influence the end-users' experience. In fact, time-critical cloud-based applications may be adopted by various users from all around the world. This type of service quality issues due to connectivity problems are currently addressed by edge computing techniques [151, 152].

- Developing a container-based auto-scaling method which is able to avoid over-provisioning of cloud-based resources is a significant challenge in the adaptation of applications. The rationale which allows host machines to locate one container instance per application type (e.g. CPU, memory, or bandwidth intensive), presented in Chapter 4, may lead to over-provisioning issue within some service clusters when some applications experience a small number of task requests. In order to overcome such a limitation, along with the container instances, host machines may be also vertically self-adjusted dynamically [153]. Another solution may be employing various host machines with respect to hardware characteristics used for each service cluster according to types of applications. For instance, hardware features of host machines which include a CPU-bound application may be different from configurations of host machines which locate a memory-intensive application. The former requires host machines with satisfactory processing power, and the later needs host machines with sufficient memory capacity.

In recent years, a wide variety of software solutions, such as IoT applications, have emerged as cloud-based systems. As a consequence, billions of users or devices get connected to applications on the Internet, which results in trillions of gigabytes of data being generated and processed in cloud datacentres. However, the burden of this large data volume, generated by end-users or devices, and transferred toward centralised cloud datacentres, leads to inefficient utilisation of resources. To overcome above problem, edge computing framework is aimed at increasing capabilities and responsibilities of resources at the edge of the network compared to traditional centralised cloud architectures by not only placing services in the proximity of end-users or devices, but also using new data transfer protocols to improve the interaction with datacentre-based services. This also provides a low-latency response time for the application.

As shown in Fig. 6.1, it is may be more efficient to process data close to the point of its source by edge nodes such as Raspberry Pi 3 model B [154], instead of remote centralised datacentres. Raspberry Pi 3 model B is a powerful credit-card sized single board computer with hardware features such as Quad Core 1.2 GHz Broadcom BCM2837 64bit CPU, 1GB RAM, 4 USB 2 ports, 4 Pole stereo output, etc.

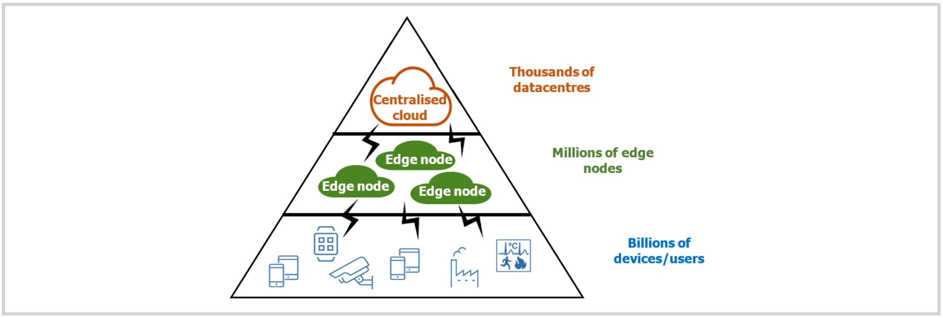


Figure 6.1

Edge computing framework.

Container-based virtualisation technology can be seen as enablers of edge computing scenarios. It is possible to deploy containerised services on edge nodes faster and more efficiently than using VMs. In essence, what makes containers an appropriate fit for edge computing scenarios is their lightweight nature. Containers provide a lightweight, portable and high performance virtualization alternative to VM-based techniques which are too heavyweight. The size of container image which is smaller than VM images makes it suitable to launch services at the edge faster than VM-based appliances. However, edge nodes allocated to run containers usually have hardware resource limitations in reality, and they are not strong enough in comparison with cloud-based hosts.

The proposed DM method can be used to run containerised services on edge nodes since edge nodes similar to cloud-based resources can be considered as infrastructures allocated to deploy containers. It should be noted that situations may arise in which an edge node is no longer able to provide computing operations since for example there are no more available computing cycles. This is because edge devices practically suffer from resource restrictions such as limited amount of computation power. In such situations, running containers deployed on the edge node should be terminated and started on the cloud side. Along this line, Fig. 6.2 shows that dynamic on/offloading action needs to occur when there is a limited amount of resources such as available computing power on the edge node at runtime.

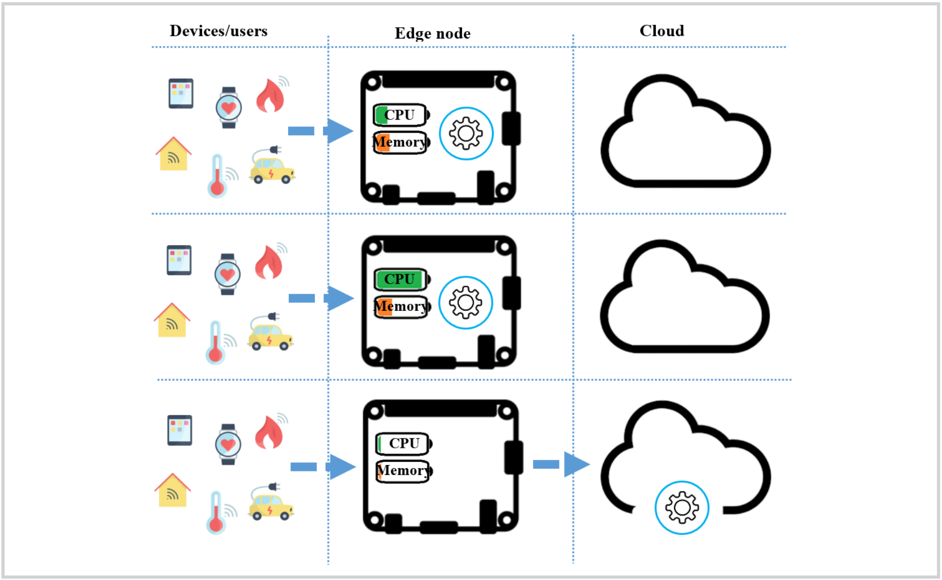


Figure 6.2

Dynamic on/offloading action within edge computing frameworks.

The proposed edge computing method continuously monitors different metrics of infrastructures (e.g. CPU, memory, etc.), and hence determines if the application performance needs to be improved at execution time. To this end, the containerised service running on the edge node can be stopped before it goes down, and then it will be launched on the cloud on-the-fly. Fig. 6.3 presents how this solution provides a dynamic deployment of containerised services—in which case each container instance is able to be used in order to accomplish specific tasks and data analytics performed at the edge. The deployment scheme to run the service on whether edge node or cloud resource can be chosen at runtime based on the application requirements.

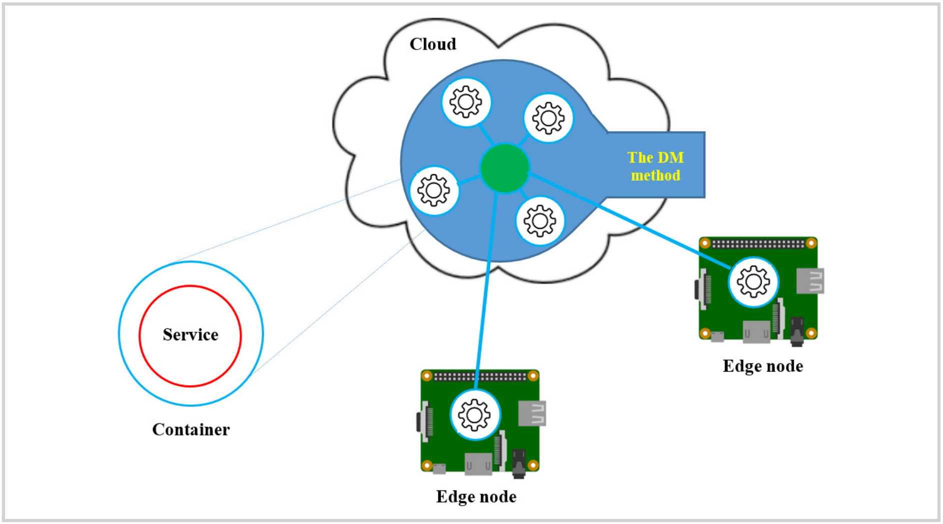


Figure 6.3

Dynamic deployment scheme within edge computing frameworks.

Therefore, an objective of such a solution is to offer a new computing mechanism to offload or onload varying workload during execution. Such a solution creates a substantial contribution to the progress of providing a dynamic, distributed architecture for an effective resource management.

Conclusion

7

This chapter provides a summary of the research performed in this thesis, and also directions for future work.

Cloud computing has become the prevalent method for providing many different types of online services over the Internet. However, auto-scaling of time-critical cloud-based applications has been a challenging issue due to runtime variations in the quantity and computational requirements of arrival requests to be processed. In essence, the main problem is that there are limited intelligent auto-scaling capabilities in these kinds of cloud environments that can be used to maintain applications' QoS requirements.

To this end, the use of a comprehensive monitoring system in the cloud is essential to track the whole range of dynamic changes in operational environments and to evaluate the performance of services offered to end-users. It helps the application provider prevent over-provisioning of resources, e.g. computing power, to predict potential issues, e.g. lack of adequate memory, and to adapt the application to avoid QoS degradation, e.g. faults at service execution time.

Primarily, most of the existing monitoring solutions do not thoroughly support the runtime auto-scaling of cloud-based applications, in order to ensure required QoS and other benefits such as optimising the resource utilisation at any time. For the purpose of bridging this gap, in our previous work [151], we focused on the comparison and analysis of the technical characteristics of cloud monitoring technologies to determine their strengths and weaknesses precisely in the domain of adaptation needs. The results of this study could also be used as a comprehensive overview of cloud monitoring approaches to come up with various self-* (self-adaptation, self-configuration, self-optimization, self-healing, self-protection, etc.) competencies needed for these applications.

Comparison of widely-used monitoring tools for cloud computing scenarios enabled us to identify monitoring requirements which are needed to support auto-scaling containerised time-critical applications. The capability of monitoring containers in this thesis is important since container-based technology was used as a new form of server virtualisation that facilitates horizontal scaling of time-critical cloud-based applications. Therefore, a monitoring system was developed to appropriately fulfil the requirements of containerised time-critical applications.

On the other hand, a fine-grained auto-scaling method is required to appropriately address highly dynamic workload scenarios in cloud-based environments. Present, traditional self-adaptation methods used for cloud-based application adopting a set of fixed auto-scaling rules unfortunately are not able to meet QoS requirements while providing optimal resource utilisation. This thesis proposed a new Dynamic Multi-level (DM) auto-scaling method which can employ dynamic rules in order to automatically whether increase or decrease the necessary number of container instances capable of accommodating varying workloads.

The presented new DM auto-scaling method innovatively employs our developed multi-level monitoring platform as the self-adaption of container-based time-critical applications need to be tuned at different levels of cloud-based operational environments that include both application level and container level. The empirical experiments clearly proved the key competence of our method which is considered as the best provisioning mechanism among eight examined auto-scaling approaches. Remarkable advantages of adopting our proposed DM method are that it properly prevents over-provisioning as well as under-provisioning of resources, while it is able to avoid the application performance issue and cost overruns at runtime.

Since we believe that our new proposed open-source DM auto-scaling method is quite mature and well-documented, it can be used in the adaptation of cloud-based services provided by giant-tech companies. This is because we, as the members of cloud community, are aware that cloud-based infrastructures are expensive whereas a significant portion of resources is not necessary to be used to address the runtime workloads. In 2017, RightScale, which is a large company selling Software as a Service (SaaS) for cloud computing management for multiple providers, conducted its 6th annual State of the Cloud Survey [155] of the latest cloud computing trends, with a focus on Infrastructure-as-a-Service (IaaS). As a result, respondents estimate 30 percent cloud-based infrastructure waste, while RightScale has measured actual waste between 30 and 45 percent. Moreover, optimizing cloud costs is the top initiative across all cloud users (53 percent) and especially among mature cloud users (64 percent).

Razširjeni povzetek

A

A.1 Raziskovalna motivacija

V zadnjih letih so se kot oblačne storitve pojavili številni časovno kritični sistemi, kot so rešitve interneta stvari (npr. sledenje vozil) in aplikacije za analizo podatkov (npr. analiza z uporabo metode končnih elementov). To je zato ker oblačne tehnologije prinašajo prednosti, kot je na primer razširljivost. Računalništvo v oblaku ponuja dinamično uporabo virov na zahtevo in s tem uporabnikom zagotavlja kakovost storitve (QoS). Omogoča dinamično zagotavljanje in odstranjevanje oblačnih virov v odziv na trenutno povpraševanje v času delovanja.

Sprememba zahtev glede delovne obremenitve aplikacij v oblaku se lahko izvede na različne načine. Na primer, novinarski kanal za oddajanje nepričakovano prejme veliko zahtevkov za obdelavo, saj se videoposnetek ali določena novica nenadoma širi v svetu socialnih medijev. Drugi primer so sistemi za paketno obdelavo podatkov, kjer se zahtevki običajno kopičijo v določenih časovnih obdobjih. Te vrste sistemov imajo običajno kratka časovna obdobja, ko je kakovost storitve zagotovljena na najnižji ravni.

Konkretna primera skalabilnih ponudnikov infrastrukture v oblaku so Amazon EC2 [1] in Microsoft Azure [2]. Le-ti lahko dinamično povečujejo ali zmanjšujejo vire dodeljene aplikacijam glede na trenutno obremenitev. Vendar pa skalabilnost teh infrastruktur ni rešitev za doseganje visoke zmogljivosti. Če tehnologije za samodejno prilagajanje, ki jih ponujajo ponudniki infrastrukture v oblaku, ne morejo zagotoviti potrebnih virov, bo kakovost aplikacij neprimerna in s tem bodo uporabniki zavrnjeni. Poleg tega bodo potencialni prihranki oziroma prednosti uporabe oblaka ogroženi, če bi uporabili več računskih virov, kot je to potrebno.

A.2 Raziskovalni problemi

Namen tega dela je odpraviti pomanjkljivosti trenutnih metod za samodejno prilagajanje aplikacij v okvirih okolij računalništva v oblaku. Zagotavljanje rezultatov visoke kakovosti pod pogojih dinamično spreminjajočih delovnih obremenitev je nujno za časovno kritične oblačne aplikacije, da bodo uporabne v poslovnem kontekstu.

Čeprav obstoječe reaktivne metode samodejnega prilagajanja uporabljajo pravila s fiksnimi pragovi, ki večinoma temeljijo na infrastrukturnih metrikah, kot sta CPU in izkoristek spomina. To vključuje reaktivne metode samodejnega prilagajanja, ki jih uporabljajo komercialni ponudniki oblačnih infrastruktur, ki temeljijo na virtualnih strojih, kot sta Microsoft Azure [2] in Amazon EC2 [1] ter tudi orkestratorji odprtokodnih vsebnikov, kot sta Kubernetes [3] in OpenShift Origin [4]. Vendar lahko take metode za samodejno prilagajanje na podlagi pravil delujejo za nekatere osnovne tipe oblačnih aplikacij, v primeru časovno kritičnih aplikacij pa beležijo poslabšanje zmogljivosti in izkoristek virov. Ti statični mehanizmi za samodejno prilagajanje aplikacij na podlagi pravil niso dovolj prožni, da se popolnoma prilagodijo obremenitvam.

Primer pravila za samodejno prilagajanje na podlagi metrike CPU, ki ga lahko določimo, je da bo več virtualnih strojev oziroma vsebnikov instanciranih, če povprečna poraba CPU doseže fiksni prag, npr. 80%; medtem, ko je mogoče nekatere virtualne stroje / vsebnike ustaviti, če je povprečna poraba CPU manjša od 80%. Takšne nastavitve ne morejo biti zelo koristne za specifične scenarije delovnih obremenitev, npr. drastično spreminjajočega vzorca. Poleg tega ta pravila za samodejno prilagajanje dosežejo v najboljšem primeru stabilno delovanje pri 80% porabi virov, kar pomeni, da je 20% virov neporabljenih, kar pa ni donosno. Eden od pomembnih izzivov in glavnih tehničnih vprašanj pri zagotavljanju metode samodejnega prilagajanja je določiti, v kolikšni meri je lahko taka metoda samodejno prilagodljiva glede na različice delovanja v obratovalnih pogojih.

Proaktivne metode samodejnega prilagajanja [6–9] se razvijajo z namenom napovedovanja količine potrebnih virov v bližnji prihodnosti na podlagi zbranih podatkov preteklega spremljanja, trenutne intenzivnosti delovnih obremenitev, kakovosti storitev aplikacije in podobno. Proaktivni pristopi za samodejno prilagajanje aplikacij uporabljajo algoritme za strojno učenje [10, 11], nevronske mreže [12, 13], teorijo čakalnih vrst [14, 15], metode podatkovnega rudarjenja [16, 17] in regresijske modele [18, 19] za napovedovanje količine potrebnih virov. Potrebno je opozoriti, da te metode zahtevajo dovolj preteklih podatkov za učenje ustreznih modelov in dovolj časa za konvergenco k stabilnemu modelu. Če imajo proaktivni načini samodejnega merjenja dovolj veliko podatkov za učenje, ki odražajo značilnosti vseh različnih možnih operativnih situacij, so sposobni posploševati in se lahko odzivajo na nove scenarije delovnih obremenitev. Posledično, če učni podatki niso dovolj izčrpni, lahko taki proaktivni pristopi trpijo zaradi njihove meje nenatančnosti, ki lahko povzroči prekomerno izkoriščanje virov ali zmanjšano zmogljivost aplikacije.

A.3 Raziskovalni cilji

Še vedno je mogoče znatno izboljšati metode samodejnega prilagajanja časovno kritičnih oblačnih aplikacij [20–25]. Metoda samodejnega prilagajanja ima za posledico stanja, kjer je uporaba dodeljenih virov nesprejemljivo nizka ali stanja podhranjenosti z viri, kjer aplikacija trpi zaradi poslabšane zmogljivosti. Zato je zahtevana dinamična metoda samodejnega prilagajanja kot odziv na dinamična nihanja delovnih obremenitev med izvajanjem aplikacij.

V času izvajanja drobnozrnata metoda samodejnega prilagajanja dinamično razporedi optimalno količino virov, potrebnih za zagotavljanje zmogljivosti aplikacije, pri čemer zagotavlja balans med porabo in zmogljivostjo. Tak pristop pri samodejnem prilagajanju mora biti sposoben izpolnjevati zahteve glede učinkovitosti aplikacij (kot so časovne omejitve odziva) in hkrati optimizirati uporabo virov, dodeljenih aplikaciji (kot je število virtualnih strojev / vsebnikov).

Iz opisanega sledijo cilji, ki jih bo morala doseči naša nova metoda za samodejno prilagajanje:

- *Izboljšanje učinkovitosti časovno kritičnih aplikacij*: prilagoditev časovno kritičnih aplikacij v oblaku, pod pogojih dinamičnega spreminjanja obremenitev med izvajanjem, je bistvenega pomena za zagotavljanje zmogljivosti aplikacij v smislu odziva na uporabniške zahteve. Oblačne aplikacije potrebujejo metode samodejnega prilagajanja, ki lahko zagotavljajo odzivnost aplikacij ob spremembah delovnih obremenitev.

- *Optimiziranje izkoriščanja virov v oblaku*: mehanizem samodejnega prilagajanja mora nenehno prilagajati optimalno količino dodeljenih virov, potrebnih za doseganje zmogljivosti časovno kritičnih oblačnih aplikacij. Sprostitev nedejavnih virov je koristno za boljšo izrabo virov v oblaku. Neizkoriščenost virov je še vedno resna težava v podatkovnih centrih [26–29]. Potrebno je povečati izkoristek virov in tako prihraniti vire in zmanjšati porabo energije.

A.4 Znanstveni prispevki

Ključne prispevke te teze je mogoče povzeti takole:

- *predstavitev novega večnivojskega sistema za nadzor časovno kritičnih oblačnih aplikacij, ki smo ga objavili pod licenco Apache 2.0*: eden od glavnih prispevkov raziskovalnega dela je obravnavanje zahtev časovno kritičnih oblačnih aplikacij, ki so pakirane v vsebnikih ali virtualnih strojih in na podlagi tega razvoj sistema za nadzor aplikacij, ki sledi celotnemu življenjskemu ciklu aplikacij, in sicer od njihovega načrtovanja, inženiringa, uvajanja in obratovalne faze. Sistem za nadzor mora odgovoriti različnim zahtevam končnih uporabnikov, operaterjev aplikacij, kot tudi podatkovnih centrov. Zato je potreben nov večnivojski sistem za nadzor, ki bi podrobneje opredelil potrebe in razmerja med različnimi metrikami kakovosti na vseh nivojih: infrastrukture, vsebnikov in virtualnih strojev ter na ravni aplikacij. V zadnjih letih so razvili veliko različnih orodij za nadzor, vendar nobeno od obstoječih orodij za nadzor ne obravnava celotnega življenjskega cikla časovno kritičnih oblačnih aplikacij.

- *predlog metode inovativnih dinamičnih pragov, uporabljenih za prilagajanje, kot odziv na spreminjajočo se delovno obremenitev v daljšem časovnem obdobju*: v zadnjih letih je veliko raziskovalnih del zagotavljalo različne načine samodejnih pravil prilagajanja, ki uporabljajo samo statične, fiksne prage. Čeprav so takšne metode, ki temeljijo na pravilih, lahko koristne za nekatere osnovne vrste delovnih obremenitev, se njihova učinkovitost in izkoriščenost virov zmanjša, ko gre za dinamično spreminjajoče se scenarije delovnih obremenitev. Eden od glavnih odprtih izzivov pri razvoju nove metode samodejnega prilagajanja z uporabo pravil je, da odločimo, v kolikšni meri bi moral biti prilagoditveni pristop samoregulativen glede na spremembe v izvajalnem okolju. Da bi naslovili ta izziv, naša nova metoda uporablja dinamične prage.

- *uvajamo fino-zrnato metodo samodejnega prilagajanja, ki omogoča ne le doseganje ugodne kakovosti storitev, temveč tudi optimalno izkoriščanje virov*: potrebujemo natančne mehanizme samodejnega prilagajanja, ki se soočajo z zelo dinamičnimi delovnimi obremenitvami v oblaku. Obstoječi tradicionalni pristopi prilagajanja aplikacij žal ne morejo natančno zagotavljati ugodne kakovosti storitev in nuditi optimalno izrabo virov. Ta teza uvaja novo metodo samodejnega skaliranja, ki uporablja podatke iz večnivojskega sistema za nadzor, in uporablja dinamična pravila za samodejno skaliranje, da se lahko aplikacija prilagodi različnim delovnim obremenitvem in zagotovi sprejemljive odzivne čase.

A.5 Rezultati

Obstajajo številne metode samodejnega prilagajanja, ki jih uporabljajo komercialni ponudniki oblakov, čeprav večinoma uporabljajo statična pravila za samodejno prilagajanje, ki niso dovolj prožna. Poleg tega obstoječe metode samodejnega prilagajanja ne uporabljajo sočasno infrastukturne metrike in metrike na ravni aplikacij za dinamično določanje števila instanc potrebnih vsebnikov ali za ustavitev delovanja nepotrebnih vsebnikov.

V tej diplomski nalogi smo predlagali novo dinamično večstopenjsko metodo (DM) z dinamičnim spreminjanjem pragov. Predlagana metoda DM uporablja ne le infrastrukturne, ampak tudi metrike na ravni aplikacij, da bi dodelila optimalno količino virov (glede na potrebno število primerkov vsebnikov), ki so potrebni za zagotovitev učinkovitosti aplikacij (npr. glede na odzivni čas storitev) brez premalo ali preveč dodeljenih virov.

Novo DM metodo smo primerjali s sedmimi obstoječimi pristopi avtomatskega prilagajanja z uporabo pravil v različnih scenarijih sintetičnih in realnih delovnih obremenitev. Te metode so: Horizontal Pod Auto-scaling (HPA), Step Scaling 1 in 2 (SS1, SS2), Target Tracking Scaling 1 in 2 (TTS1, TTS2) ter Static Threshold Based Scaling 1 in 2 (THRES1, THRES2). Predlagana metoda DM v primerjavi z drugimi metodami dosega najboljše splošne rezultate glede na izkoristek virov in zagotavljanja odzivnih časov. V zvezi z vsemi scenariji delovnih obremenitev, preučevanimi v tem delu (predstavljajo vzorce vztrajno naraščajočih/padajočih, drastično spreminjajočih, rahlih sprememb ter dejanskih delovnih obremenitev), je temeljna prednost nove DM metode njena sposobnost uporabiti večnivojski sistem za nadzor nad izvajanjem aplikacij ter se dinamično prilagajati delovnim obremenitvam.

A.6 Zaključek

Računalništvo v oblaku je postalo prevladujoči način za zagotavljanje številnih spletnih storitev prek medmrežja. Vendar pa predstavlja samodejno prilagajanje časovno kritičnih aplikacij težavo zaradi sprememb v količini in računskih zahtevnosti prihajajočih zahtevkov, ki jih je potrebno obdelati.

V ta namen je uporaba celovitega sistema za nadzor oblačnih aplikacij bistvenega pomena za sledenje celotnemu obsegu dinamičnih sprememb v izvajalnem okolju in za ocenjevanje kakovosti storitev, ponujenih končnim uporabnikom. Ponudniku aplikacij pomaga preprečiti prevelik zajem virov, npr. računalniško moč, predvidi morebitne težave, npr. pomanjkanja ustreznega pomnilnika in omogoča prilagajanje aplikacij, da se izogne poslabšanju kakovosti, npr. prevelikih odzivnih časov.

Primerjava široko uporabljenih orodij za nadzor nad izvajanjem aplikacij nam je omogočila analizo osnovnih zahtev za nadzor, ki so zahtevane pri samodejnem prilagajanju časovno kritičnih aplikacij. Zmožnost spremljanja vsebnikov v tej diplomski nalogi je pomembna, saj je tehnologija, ki temelji na vsebnikih, uporabljena kot nova oblika virtualizacije, ki omogoča horizontalno skaliranje časovno kritičnih oblačnih aplikacij. Zato je bil razvit večnivojski sistem za nadzor, da bi zagotovili ustrezne pogoje za izvajanje časovno kritičnih aplikacij.

Po drugi strani pa je potrebna natančna metoda za nadzor, ki ustreza zelo dinamičnim scenarijem delovne obremenitve v oblačnih okoljih. Sedanji tradicionalni načini samodejnega prilagajanja, uporabljeni za oblačne aplikacije, ki sprejme niz določenih pravil samodejnega merjenja, na žalost ne morejo zadovoljiti zahtev QoS ob zagotavljanju optimalne uporabe virov. Ta teza predlaga novo metodo dinamičnega večnivojskega (DM) samodejnega skaliranja, ki uporablja dinamična pravila, samodejno povečuje ali zmanjšuje potrebno število primerkov vsebnikov in tako zagotavlja ustrezno kakovost ob različnih delovnih obremenitvah.

Predstavljena nova metoda avtomatskega prilagajanja DM inovativno uporablja naš razviti večnivojski sistem za nadzor časovno kritičnih aplikacij, ki temeljijo na vsebnikih, z namenom prilagajanja različnim nivojem v izvajalnem okolju, ki vključujejo raven aplikacije kot tudi raven vsebnika. Empirični poskusi so jasno pokazali ključne prednosti naše metode, ki trenutno šteje kot najboljši mehanizem zagotavljanja virov med osmimi preučenimi pristopi samodejnega prilagajanja z uporabo pravil. Izredne prednosti ob uvajanju predlagane DM metode so, da pravilno preprečuje preveliko in premajhno dodeljevanje virov, hkrati pa se izogiba težavam pri izvajanju aplikacij in prekoračitvi obratovalnih stroškov.

Abbreviations

B

Abbreviations

ACCRS	Autonomic Cloud Computing Resource Scaling
ARMA	Autoregressive-Moving Average
ARNES	Academic and Research Network of Slovenia
ASF	Apache Software Foundation
ATLB	Application Throughput of the Load-Balancer
CDF	Cumulative Distribution Function
CPU	Central Processing Unit
CQL	Cassandra Query Language
DC2	Dependable Compute Cloud
DM	Dynamic Multi-Level
DUCP	Docker Universal Control Plane
ECS	Amazon EC2 Container Service
ESB	Event Service Bus
FQL	Fuzzy Q-learning Learning
FSL	Fuzzy SARSA Learning
GKE	Google Container Engine
GUI	Graphical User Interface
HPA	Horizontal Pod Auto-scaling
IaaS	Infrastructure-as-a-Service
IDE	Interactive Development Environment
IoT	Internet of Things

JVM	Java Virtual Machine
KB	Knowledge Base
MDP	Markov Decision Process
MP	Multiple Policies
NoP	Number of Pods
QoS	Quality of Service
RDF	Resource Description Framework
RPS	Requests Per Second
RT	Response Time
SaaS	Software as a Service
SLA	Service Level Agreement
SNMP	Simple Network Management Protocol
SQL	Structured Query Language
SS	Step Scaling
TSDB	Time Series Database
TTS	Target Tracking Scaling
VM	Virtual Machine
XDR	External Data Representation
XML	Extensible Markup Language

BIBLIOGRAPHY

[1] *Amazon EC2*, 2018 (Accessed April 15, 2014). URL https://aws.amazon.com/ec2.

[2] *Microsoft Azure*, 2018 (Accessed April 15, 2014). URL https://azure.microsoft.com/.

[3] *Kubernetes*, 2018 (Accessed April 15, 2014). URL https://kubernetes.io/.

[4] *OpenShift Origin*, 2018 (Accessed April 15, 2014). URL https://www.openshift.org/.

[5] Z. Zhao, P. Martin, J. Wang, A. Taal, A. Jones, I. Taylor, V. Stankovski, I.G. Vega, G. Suciu, A. Ulisses, and C. de Laata. Developing and operating time critical applications in clouds: the state of the art and the switch approach. 68:17–28, 2015. doi: https://doi.org/10.1016/j.procs.2015.09.220.

[6] M. Ghobaei-Arani, S. Jabbehdari, and M.A. Pourmina. An autonomic resource provisioning approach for service-based cloud applications: a hybrid approach. *Future Generation Computer Systems*, 78:191–210, 2018. doi: https://doi.org/10.1016/j.future.2017.02.022.

[7] M.S. Aslanpour and S.E. Dashti. Proactive auto-scaling algorithm (pasa) for cloud application. *International Journal of Grid and High Performance Computing (IJGHPC)*, 9(3):1–16, 2017.

[8] Q. Zhang, H. Chen, and Z. Yin. Prmrap: A proactive virtual resource management framework in cloud. In *Proc. of 2017 IEEE International Conference on Edge Computing (EDGE)*, pages 120–127, Honolulu, HI, USA, 2017. IEEE. doi: 10.1109/IEEE.EDGE.2017.24.

[9] D. Tran, N. Tran, G. Nguyen, and B.M. Nguyen. A proactive cloud scaling model based on fuzzy time series and sla awareness. *Procedia Computer Science*, 108:365–374, 2017. doi: https://doi.org/10.1016/j.procs.2017.05.121.

[10] J.V.B. Benifa and D. Dejey. Rlpas: Reinforcement learning-based proactive auto-scaler for resource provisioning in cloud environment. *Mobile Networks and Applications*, pages 1–16, 2018. doi: https://doi.org/10.1007/s11036-018-0996-0.

[11] J. Rao, X. Bu, C.Z. Xu, and K. Wang. A distributed self-learning approach for elastic provisioning of virtualized cloud resources. In *Proc. of 2011 IEEE 19th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 45–54, Singapore, Singapore, 2011. IEEE. doi: 10.1109/MASCOTS.2011.47.

[12] J. Kumar and A.K. Singh. Workload prediction in cloud using artificial neural network and adaptive differential evolution. *Future Generation Computer Systems*, 81:41–52, 2018. doi: https://doi.org/10.1016/j.future.2017.10.047.

[13] T. Chen and R. Bahsoon. Self-adaptive and sensitivity-aware qos modeling for the cloud. In *Proc. of 2013 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 43–52, San Francisco, CA, USA, 2013. IEEE. doi: 10.1109/SEAMS.2013.6595491.

[14] B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal, and T. Wood. Agile dynamic provisioning of multi-tier internet applications. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 3(1):1, 2008. doi: 10.1145/1342171.1342172.

[15] R. Han, M.M. Ghanem, L. Guo, Y. Guo, and M. Osmond. Enabling cost-aware and adaptive elasticity of multi-tier cloud applications. *Future Generation Computer Systems*, 32:82–98, 2014. doi: https://doi.org/10.1016/j.future.2012.05.018.

[16] E. Yuan, N. Esfahani, and S. Malek. Automated mining of software component interactions for self-adaptation. In *Proc. of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 27–36, Hyderabad, India, 2014. ACM. doi: 10.1145/2593929.2593934.

[17] K. Wac, A.V. Halteren, and D. Konstantas. Qos-predictions service: Infrastructural support for proactive qos-and context-aware mobile services (position paper). In *Proc. of OTM 2006 Workshops On the Move to Meaningful Internet Systems*, pages 1924–1933, Berlin, Germany, 2006. Springer. doi: http://dx.doi.org/10.1007/11915072_109.

[18] A. Gambi and G. Toffetti. Modeling cloud performance with kriging. In *Proc. of 2012 34th International Conference on Software Engineering (ICSE)*, pages 1439–1440, Zurich, Switzerland, 2012. IEEE. doi: 10.1109/ICSE.2012.6227075.

[19] Q. Zhu and G. Agrawal. Resource provisioning with budget constraints for adaptive applications in cloud environments. volume 5, pages 497–511. IEEE, 2012. doi: 10.1109/TSC.2011.61.

[20] N. Anwar and H. Deng. Elastic scheduling of scientific workflows under deadline constraints in cloud computing environments. *Future Internet*, 10(1):5, 2018. doi: 10.3390/fi10010005.

[21] P.C. Amogh, G. Veeramachaneni, A.K. Rangisetti, B.R. Tamma, and A.A. Franklin. A cloud native solution for dynamic auto scaling of mme in lte. In *Proc. of 2017 IEEE 28th Annual International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC)*, pages 1–7, Montreal, QC, Canada, 2017. IEEE. doi: 10.1109/PIMRC.2017.8292270.

[22] A.S. Prasad, D. Koll, J.O. Iglesias, J.A. Aroca, V. Hilt, and X. Fu. Rconf (pd): Automated resource configuration of complex services in the cloud. *Future Generation Computer Systems*, 2018. doi: https://doi.org/10.1016/j.future.2018.02.027.

[23] Z. Cai, X. Li, and R. Ruiz. Resource provisioning for task-batch based workflows with deadlines in public clouds. *IEEE Transactions on Cloud Computing*, 2017. doi: 10.1109/TCC.2017.2663426.

[24] G. Suciu, A. Scheianu, and M. Vochin. Disaster early warning using time-critical iot on elastic cloud workbench. In *Proc. of 2017 IEEE International Black Sea Conference on Communications and Networking (BlackSeaCom)*, pages 1–5, Istanbul, Turkey, 2017. IEEE. doi: 10.1109/BlackSeaCom.2017.8277712.

[25] J. Wang, A. Taal, P. Martin, Y. Hu, H. Zhou, J. Pang, C. de Laat, and Z. Zhao. Planning virtual infrastructures for time critical applications with multiple deadline constraints. *Future Generation Computer Systems*, 75:365–375, 2017. doi: https://doi.org/10.1016/j.future.2017.02.001.

[26] M.S. Aslanpour, M. Ghobaei-Arani, and A.N. Toosi. Auto-scaling web applications in clouds: a cost-aware approach. *Journal of Network and Computer Applications*, 95:26–41, 2017.

[27] M.S. Aslanpour and S.E. Dashti. Sla-aware resource allocation for application service providers in the cloud. In *Proc. of 2016 Second International Conference on Web Research (ICWR)*, pages 31–42. IEEE, 2016. doi: 10.1109/ICWR.2016.7498443.

[28] A.A. Rahmanian, G.H. Dastghaibyfard, and H. Tahayori. Penalty-aware and cost-efficient resource management in cloud data centers. *International Journal of Communication Systems*, 30(8), 2017.

[29] D. Moldovan, H.L. Truong, and S. Dustdar. Cost-aware scalability of applications in public clouds. In *Proc. of 2016 IEEE International Conference on Cloud Engineering (IC2E)*, pages 79–88, Berlin, Germany, 2016. IEEE. doi: 10.1109/IC2E.2016.23.

[30] V.V. Trofimov, V.I. Kiyaev, and S.M. Gazul. Use of virtualization and container technology for information infrastructure generation. In *Proc. of 2017 IEEE International Conference on Soft Computing and Measurements (SCM)*, pages 788–791, St. Petersburg, Russia, 2017. IEEE. doi: 10.1109/SCM.2017.7970725.

[31] S. Taherizadeh and V. Stankovski. Auto-scaling applications in edge computing: Taxonomy and challenges. In *Proc. of the International Conference on Big Data and Internet of Thing*, pages 158–163, London, United Kingdom, 2017. ACM. doi: 10.1145/3175684.3175709.

[32] *Docker*, 2018 (Accessed April 15, 2014). URL `https://www.docker.com/`.

[33] A. Patel, M. Daftedar, M. Shalan, and M.W. El-Kharashi. Embedded hypervisor xvisor: A comparative analysis. In *Proc. of 2015 23rd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 682–691, Turku, Finland, 2015. IEEE. doi: 10.1109/PDP.2015.108.

[34] *Google container engine*, 2018 (Accessed April 15, 2014). URL `https://cloud.google.com/container-engine/`.

[35] *Amazon EC2 Container Service*, 2018 (Accessed April 15, 2014). URL `https://aws.amazon.com/ecs/`.

[36] I. Emsley and D. De-Roure. A framework for the preservation of a docker container. *The International Journal of Digital Curation (IJDC)*, 12(2):125–135, 2017. doi: https://doi.org/10.2218/ijdc.v12i2.509.

[37] S.P. Polyakov, A.P. Kryukov, and A.P. Demichev. Docker container manager: A simple toolkit for isolated work with shared computational, storage, and network resources. In *Journal of Physics: Conference Series*, volume 955, pages 1–4. IOP Publishing, 2018. doi: 10.1088/1742-6596/955/1/012039.

[38] *Docker Hub repository*, 2018 (Accessed April 15, 2014). URL `https://hub.docker.com/`.

[39] *Pushing and Pulling Docker Images*, 2018 (Accessed April 15, 2014). URL `https://cloud.google.com/container-registry/docs/pushing-and-pulling`.

[40] *Dockerfile reference*, 2018 (Accessed April 15, 2014). URL `https://docs.docker.com/engine/reference/builder/`.

[41] *Developed monitoring system*, 2018 (Accessed April 15, 2014). URL https://github.com/salmant/ASAP/tree/master/SWITCH-Monitoring-System.

[42] G. Aceto, A. Botta, W. De-Donato, and A. Pescape. Cloud monitoring: A survey. *Computer Networks*, 57(9):2093–2115, 2013. doi: https://doi.org/10.1016/j.comnet.2013.04.001.

[43] G. Aceto, A. Botta, W. De-Donato, and A. Pescape. Cloud monitoring: definitions, issues and future directions. In *Proc. of 2012 IEEE 1st International Conference on Cloud Networking (CLOUDNET)*, pages 63–67, Paris, France, 2012. IEEE. doi: 10.1109/CloudNet.2012.6483656.

[44] K. Fatema, V.C. Emeakaroha, P.D. Healy, J.P. Morrison, and T. Lynn. A survey of cloud monitoring tools: Taxonomy, capabilities and objectives. *Journal of Parallel and Distributed Computing*, 74(10):2918–2933, 2014. doi: https://doi.org/10.1016/j.jpdc.2014.06.007.

[45] M. García-Valls, T. Cucinotta, and C. Lu. Challenges in real-time virtualization and predictable cloud computing. *Journal of Systems Architecture*, 60(9):726–740, 2014. doi: https://doi.org/10.1016/j.sysarc.2014.07.004.

[46] M.H. Mohamaddiah, A. Abdullah, S. Subramaniam, and M. Hussin. A survey on resource allocation and monitoring in cloud computing. *International Journal of Machine Learning and Computing*, 4(1):31, 2014. doi: 10.7763/IJMLC.2014.V4.382.

[47] J.S. Ward and A. Barker. Observing the clouds: a survey and taxonomy of cloud monitoring. *Journal of Cloud Computing*, 3(1):24, 2014. doi: https://doi.org/10.1186/s13677-014-0024-2.

[48] B. Hazarika and T. Sing. Survey paper on cloud computing and cloud monitoring: basics. *SSRG Int. J. Comput. Sci. Eng*, 2(1):10–15, 2015.

[49] J.M.A. Calero and J.G. Aguado. Comparative analysis of architectures for monitoring cloud computing infrastructures. *Future Generation Computer Systems*, 47:16–30, 2015. doi: https://doi.org/10.1016/j.future.2014.12.008.

[50] K. Sugapriya and J.S. Jeya. A survey on enhanced scheme for multiple cloud resource matchmaking using trust aware framework. *International Journal of Advanced Research in Computer Science and Software Engineering*, 5(11):1–4, 2015.

[51] K. Alhamazani, R. Ranjan, K. Mitra, F. Rabhi, P.P. Jayaraman, S.U. Khan, A. Guabtni, and V. Bhatnagar. An overview of the commercial cloud monitoring tools: research dimensions, design issues, and state-of-the-art. *Computing*, 97(4):357–377, 2015. doi: 10.1007/s00607-014-0398-5.

[52] S. Taherizadeh and V. Stankovski. Incremental learning from multi-level monitoring data and its application to component based software engineering. In *Proc. of 2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*, pages 378–383, Turin, Italy, 2017. IEEE. doi: 10.1109/COMPSAC.2017.148.

[53] *Apache Software Foundation*, 2018 (Accessed April 15, 2014). URL https://www.apache.org/.

[54] S. Clayman, A. Galis, and L. Mamatas. Monitoring virtual networks with lattice. In *Proc. of 2010 IEEE/IFIP Network operations and management symposium workshops (NOMS Wksps)*, pages 239–246, Osaka, Japan, 2010. IEEE. doi: 10.1109/NOMSW.2010.5486569.

[55] *cAdvisor (Container Advisor)*, 2018 (Accessed April 15, 2014). URL https://github.com/google/cadvisor.

[56] Edward Campbell. *Apache Mesos Basics*. CreateSpace Independent Publishing Platform, USA, 2017. ISBN 1548267635, 9781548267636.

[57] *Docker Swarm*, 2018 (Accessed April 15, 2014). URL https://docs.docker.com/swarm/.

[58] *Docker Remote API*, 2018 (Accessed April 15, 2014). URL https://docs.docker.com/engine/api/v1.21/.

[59] *InfluxDB*, 2018 (Accessed April 15, 2014). URL https://influxdata.com/time-series-platform/influxdb/.

[60] *Elasticsearch*, 2018 (Accessed April 15, 2014). URL https://www.elastic.co/blog/elasticsearch-as-a-time-series-data-store.

[61] *BigQuery*, 2018 (Accessed April 15, 2014). URL https://cloud.google.com/bigquery/.

[62] *Grafana*, 2018 (Accessed April 15, 2014). URL http://grafana.org/.

[63] *Prometheues*, 2018 (Accessed April 15, 2014). URL https://prometheus.io/.

[64] R. Peinl, F. Holzschuher, and F. Pfitzer. Docker cluster management for the cloud-survey results and own solution. *Journal of Grid Computing*, 14(2):265–282, 2016. doi: 10.1007/s10723-016-9366-y.

[65] *Docker Universal Control Plane (DUCP)*, 2018 (Accessed April 15, 2014). URL https://docs.docker.com/ucp/.

[66] *Scout*, 2018 (Accessed April 15, 2014). URL https://scoutapp.com/.

[67] S. Taherizadeh and V. Stankovski. Quality of service assurance for internet of things time-critical cloud applications: Experience with the switch and entice projects. In *Proc. of 2017 6th IIAI International Congress on Advanced Applied Informatics (IIAI-AAI)*, pages 288–293, Hamamatsu, Japan, 2017. IEEE. doi: 10.1109/IIAI-AAI.2017.209.

[68] *Zenoss*, 2018 (Accessed April 15, 2014). URL `http://www.zenoss.org/`.

[69] U. Gupta. Monitoring in iot enabled devices. *arXiv preprint arXiv:1507.03780*, 2015.

[70] *Ganglia*, 2018 (Accessed April 15, 2014). URL `http://ganglia.info/`.

[71] M.L. Massie, B.N. Chun, and D.E. Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817–840, 2004. doi: https://doi.org/10.1016/j.parco.2004.04.001.

[72] *sFlow*, 2018 (Accessed April 15, 2014). URL `http://blog.sflow.com/2010/10/ganglia.html`.

[73] *RRDtool*, 2018 (Accessed April 15, 2014). URL `https://oss.oetiker.ch/rrdtool/`.

[74] *Zabbix*, 2018 (Accessed April 15, 2014). URL `http://www.zabbix.com/`.

[75] P. Tader. Server monitoring with zabbix. *Linux Journal*, 2010(195):7, 2010.

[76] J.W. Murphy. *SnoScan: An iterative functionality service scanner for large scale networks*. Iowa State University, 2008.

[77] S. Clayman, A. Galis, C. Chapman, G. Toffetti, L. Rodero-Merino, L.M. Vaquero, K. Nagin, and B. Rochwerger. Monitoring service clouds in the future internet. In *Towards the Future Internet - Emerging Trends from European Research*, pages 115–126. IOS Press, 2010. doi: 10.3233/978-1-60750-539-6-115.

[78] G. Katsaros, R. Kubert, G. Gallizo, and T. Wang. Monitoring: A fundamental process to provide qos guarantees in cloud-based platforms. *Cloud Computing*, pages 325–341, 2011.

[79] D. Trihinas, G. Pallis, and M.D. Dikaiakos. Jcatascopia: Monitoring elastically adaptive applications in the cloud. In *Proc. of 2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 226–235, Chicago, IL, USA, 2014. IEEE. doi: 10.1109/CCGrid.2014.41.

[80] *StatsD protocol*, 2018 (Accessed April 15, 2014). URL `https://github.com/etsy/statsd/wiki`.

[81] *Developed Monitoring Server container image*, 2018 (Accessed April 15, 2014). URL `https://hub.docker.com/r/salmant/ul_monitoring_server_container_image/`.

[82] T. Lorido-Botran, J. Miguel-Alonso, and J.A. Lozano. A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of grid computing*, 12(4):559–592, 2014.

[83] L. Baresi, S. Guinea, A. Leva, and G. Quattrocchi. A discrete-time feedback controller for containerized cloud applications. In *Proc. of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 217–228, Seattle, WA, USA, 2016. ACM. doi: 10.1145/2950290.2950328.

[84] 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.

[85] 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.

[86] P. Jamshidi, A.M. Sharifloo, C. Pahl, A. Metzger, and G. Estrada. Self-learning cloud controllers: Fuzzy q-learning for knowledge evolution. In *Proc. of International Conference on Cloud and Autonomic Computing (ICCAC)*, pages 208–211, Boston, USA, 2015. IEEE. doi: 10.1109/ICCAC.2015.35.

[87] H. Arabnejad, C. Pahl, P. Jamshidi, and G. Estrada. A comparison of reinforcement learning techniques for fuzzy cloud auto-scaling. In *Proc. of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 64–73, Madrid, Spain, 2017. IEEE. doi: 10.1109/CCGRID.2017.15.

[88] D. Tsoumakos, I. Konstantinou, C. Boumpouka, S. Sioutas, and N. Koziris. Automated, elastic resource provisioning for nosql clusters using tiramola. In *Proc. of 2013 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 34–41, Delft, Netherlands, 2013. IEEE. doi: 10.1109/CCGrid.2013.45.

[89] A. Gandhi, P. Dube, A. Karve, A. Kochut, and L. Zhang. Adaptive, model-driven autoscaling for cloud applications. In *Proc. of the 11th International Conference on Autonomic Computing (ICAC'14)*, volume 14, pages 57–64, Philadelphia, PA, 2014. USENIX.

[90] D. Jiang, G. Pierre, and C.H. Chi. Autonomous resource provisioning for multi-service web applications. In *Proceedings of the 19th international conference on World wide web*, pages 471–480. ACM, 2010.

[91] N.R. Herbst, S. Kounev, and R. Reussner. Elasticity in cloud computing: What it is, and what it is not. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*, pages 23–27. USENIX, 2013.

[92] Z. Al-Sharif, Y. Jararweh, A. Al-Dahoud, and L.M. Alawneh. Accrs: autonomic based cloud computing resource scaling. *Cluster Computing*, 20(3):2479–2488, 2017. doi: 10.1007/s10586-016-0682-6.

[93] S. Islam, J. Keung, K. Lee, and A. Liu. Empirical prediction models for adaptive resource provisioning in the cloud. *Future Generation Computer Systems*, 28(1):155–162, 2012. doi: https://doi.org/10.1016/j.future.2011.05.027.

[94] A. Bauer, N. Herbst, and S. Kounev. Design and evaluation of a proactive, application-aware autoscaler: Tutorial paper. In *Proc. of the 8th ACM/SPEC on International Conference on Performance Engineering*, pages 425–428, L-Aquila, Italy, 2017. ACM. doi: 10.1145/3030207.3053678.

[95] A.Y. Nikravesh, S.A. Ajila, and C.H. Lung. Towards an autonomic auto-scaling prediction system for cloud resource provisioning. In *Proc. of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 35–45, Florence, Italy, 2015. IEEE Press. doi: 10.1109/SEAMS.2015.22.

[96] L. Aniello, S. Bonomi, F. Lombardi, A. Zelli, and R. Baldoni. An architecture for automatic scaling of replicated services. In *Networked Systems*, pages 122–137. Springer, 2014.

[97] J. Loff and J. Garcia. Vadara: Predictive elasticity for cloud applications. In *Proc. of 2014 IEEE 6th International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 541–546, Singapore, Singapore, 2014. IEEE. doi: 10.1109/CloudCom.2014.161.

[98] P.P. Kukade and G. Kale. Auto-scaling of micro-services using containerization. *International Journal of Science and Research (IJSR)*, 4(9):1960–1963, 2015.

[99] C. Kan. Docloud: An elastic cloud platform for web applications based on docker. In *Proc. of 2016 18th International Conference on Advanced Communication Technology (ICACT)*, pages 478–483, Pyeongchang, South Korea, 2016. IEEE. doi: 10.1109/ICACT.2016.7423439.

[100] C. Qu, R.N. Calheiros, and R. Buyya. A reliable and cost-efficient auto-scaling system for web applications using heterogeneous spot instances. *Journal of Network and Computer Applications*, 65:167–180, 2016.

[101] P. Dube, A. Gandhi, A. Karve, A. Kochut, and L. Zhang. Scaling a cloud infrastructure, 2016. US Patent 9,300,553.

[102] *Kubernetes Horizontal Pod Auto-scaling*, 2018 (Accessed April 15, 2014). URL `https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/`.

[103] *Amazon Target Tracking Scaling*, 2018 (Accessed April 15, 2014). URL `http://docs.aws.amazon.com/autoscaling/latest/userguide/as-scaling-target-tracking.html`.

[104] *Amazon Step Scaling*, 2018 (Accessed April 15, 2014). URL `https://aws.amazon.com/blogs/aws/auto-scaling-update-new-scaling-\policies-for-more-responsive-scaling/`.

[105] *Google Multiple Policies Scaling*, 2018 (Accessed April 15, 2014). URL `https://cloud.google.com/compute/docs/autoscaler/multiple-policies`.

[106] *HAProxy*, 2018 (Accessed April 15, 2014). URL `http://www.haproxy.org`.

[107] C. Qu, R.N. Calheiros, and R. Buyya. Mitigating impact of short-term overload on multi-cloud web applications through geographical load balancing. *Concurrency and Computation: Practice and Experience*, 29 (12), 2017. doi: https://doi.org/10.1002/cpe.4126.

[108] S. Nadgowda, S. Suneja, and C. Isci. Paracloud: bringing application insight into cloud operations. In *Proc. of the 9th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 17)*, Santa Clara, California, 2017. USENIX Association.

[109] A.N. Toosi, C. Qu, M.D. de Assunção, and R. Buyya. Renewable-aware geographical load balancing of web applications for sustainable data centers. *Journal of Network and Computer Applications*, 83:155–168, 2017. doi: https://doi.org/10.1016/j.jnca.2017.01.036.

[110] N. Grozev and R. Buyya. Dynamic selection of virtual machines for application servers in cloud environments. In *Research Advances in Cloud Computing*, pages 187–210. Springer, 2017.

[111] H. Chen, Q. Wang, B. Palanisamy, and P. Xiong. Dcm: Dynamic concurrency management for scaling n-tier applications in cloud. In *Proc. of 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 2097–2104, Atlanta, GA, USA, 2017. IEEE. doi: 10.1109/ICDCS.2017.22.

[112] J. Kampars and K. Pinka. Auto-scaling and adjustment platform for cloud-based systems. In *Proc. of the 11th International Scientific and Practical Conference*, volume 2, pages 52–57, Rezekne, Latvia, 2017. doi: 10.17770/etr2017vol2.2591.

[113] A. Sangpetch, O. Sangpetch, N. Juangmarisakul, and S. Warodom. Thoth: Automatic resource management with machine learning for container-based cloud platform. 2017. doi: 10.5220/0006254601030111.

[114] V. Singh and S.K. Peddoju. Container-based microservice architecture for cloud applications. In *Proc. of 2017 International Conference on Computing, Communication and Automation (ICCCA)*, pages 847–852, Greater Noida, India, 2017. IEEE. doi: 10.1109/CCAA.2017.8229914.

[115] V. Nitu, B. Teabe, L. Fopa, A. Tchana, and D. Hagimont. Stopgap: elastic vms to enhance server consolidation. *Software: Practice and Experience*, 47(11):1501–1519, 2017. doi: https://doi.org/10.1002/spe.2482.

[116] M. Wajahat, A. Karve, A. Kochut, and A. Gandhi. Mlscale: A machine learning based application-agnostic autoscaler. *Sustainable Computing: Informatics and Systems*, 2017. doi: https://doi.org/10.1016/j.suscom.2017.10.003.

[117] X. Xu, H. Jin, S. Wu, X. Wu, and Y. Li. Fama: A middleware for fast deploying and auto scaling towards multitier applications in clouds. *Journal of Internet Technology*, 16(6):987–997, 2015. doi: 10.6138/JIT.2015.16.6.20130506.

[118] *GoDaddy*, 2018 (Accessed April 15, 2014). URL `https://www.godaddy.com/`.

[119] *GitHub*, 2018 (Accessed April 15, 2014). URL `https://github.com/`.

[120] *Stack Overflow*, 2018 (Accessed April 15, 2014). URL `https://stackoverflow.com/`.

[121] *Reddit*, 2018 (Accessed April 15, 2014). URL `https://www.reddit.com/`.

[122] *Speedtest*, 2018 (Accessed April 15, 2014). URL `http://www.speedtest.net/`.

[123] *Bitbucket*, 2018 (Accessed April 15, 2014). URL `https://bitbucket.org/`.

[124] *The Reliable, High Performance TCP/HTTP Load Balancer*, 2018 (Accessed April 15, 2014). URL `http://www.haproxy.org/they-use-it.html`.

[125] *W3C*, 2018 (Accessed April 15, 2014). URL `https://www.w3.org/`.

[126] *AWS OpsWorks*, 2018 (Accessed April 15, 2014). URL `https://aws.amazon.com/opsworks/`.

[127] *Apache Jena Fuseki*, 2018 (Accessed April 15, 2014). URL `https://jena.apache.org/documentation/fuseki2/index.html`.

[128] C. Roda, E. Navarro, and C.E. Cuesta. A comparative analysis of linked data tools to support architectural knowledge. pages 399–406, 2014.

[129] M.M. Ghanem R. Han, L. Guo and Y. Guo. Lightweight resource scaling for cloud applications. In *Proc. of 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 644–651, Ottawa, ON, Canada, 2012. IEEE. doi: 10.1109/CCGrid.2012.52.

[130] Zhihui Lu, Jie Wu, Jie Bao, and Patrick CK Hung. Ocrem: Openstack-based cloud datacentre resource monitoring and management scheme. *International Journal of High Performance Computing and Networking*, 9(1-2):31–44, 2016. doi: https://doi.org/10.1504/IJHPCN.2016.074656.

[131] G. Singhi and D. Tiwari. A load balancing approach for increasing the resource utilisation by minimizing the number of active servers. *International Journal of Computer Security and Source Code Analysis*, 3(1):11–15, 2017.

[132] M.A.H. Monil and R.M. Rahman. Implementation of modified overload detection technique with vm selection strategies based on heuristics and migration control. In *Proc. of 2015 IEEE/ACIS 14th International Conference on Computer and Information Science (ICIS)*, pages 223–227, Las Vegas, NV, USA, 2015. IEEE. doi: 10.1109/ICIS.2015.7166597.

[133] A. Alonso, I. Aguado, J. Salvachua, and P. Rodríguez. A metric to estimate resource use in cloud-based videoconferencing distributed systems. In *Proc. of 2016 IEEE 4th International Conference on Future Internet of Things and Cloud (FiCloud)*, pages 25–32, Vienna, Austria, 2016. IEEE. doi: 10.1109/FiCloud.2016.12.

[134] A. Alonso, I. Aguado, J. Salvachua, and P. Rodríguez. A methodology for designing and evaluating cloud scheduling strategies in distributed videoconferencing systems. *IEEE Transactions on Multimedia*, 19(10):2282–2292, 2017. doi: 10.1109/TMM.2017.2733301.

[135] *Developed Alarm-Trigger component*, 2018 (Accessed April 15, 2014). URL `https://github.com/salmant/ASAP/tree/master/SWITCH-Alarm-Trigger`.

[136] *The Alarm-Trigger container image*, 2018 (Accessed April 15, 2014). URL `https://hub.docker.com/r/salmant/ul_alarm_trigger_container_image/`.

[137] *Developed Self-Adapter component*, 2018 (Accessed April 15, 2014). URL `https://github.com/salmant/ASAP/tree/master/SWITCH-Self-Adapter`.

[138] *httperf*, 2018 (Accessed April 15, 2014). URL `https://github.com/httperf/httperf`.

[139] J. Juzna, P. Cesarek, D. Petcu, and V. Stankovski. Solving solid and fluid mechanics problems in the cloud with mosaic. *Computing in Science and Engineering*, 16(3):68–77, 2014. doi: 10.1109/MCSE.2013.135.

[140] *1998 World Cup Web Site Access Logs*, 2018 (Accessed April 15, 2014). URL http://ita.ee.lbl.gov/html/contrib/WorldCup.html.

[141] K. Rattanaopas and P. Tandayya. Adaptive workload prediction for cloud-based server infrastructures. *Journal of Telecommunication, Electronic and Computer Engineering (JTEC)*, 9(2-4):129–134, 2017.

[142] L. Jiao, A.M. Tulino, J. Llorca, Y. Jin, and A. Sala. Smoothed online resource allocation in multi-tier distributed cloud networks. *IEEE/ACM Transactions on Networking (TON)*, 25(4):2556–2570, 2017. doi: 10.1109/TNET.2017.2707142.

[143] D. Tran, N. Tran, B.M. Nguyen, and H. Le. Pdgabp—a novel prediction model applying for elastic applications in distributed environment. In *Proc. of 2016 3rd National Foundation for Science and Technology Development Conference on Information and Computer Science (NICS)*, pages 240–245, Danang, Vietnam, 2016. IEEE. doi: 10.1109/NICS.2016.7725658.

[144] L. Logeswaran, H.M.N. Dilum-Bandara, and H.S. Bhathiya. Performance, resource, and cost aware resource provisioning in the cloud. In *Proc. of 2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*, pages 913–916, San Francisco, CA, USA, 2016. IEEE. doi: 10.1109/CLOUD.2016.0135.

[145] L.R. Moore, K. Bean, and T. Ellahi. A coordinated reactive and predictive approach to cloud elasticity. 2013.

[146] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle. Autonomic vertical elasticity of docker containers with elasticdocker. In *Proc. of 2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*, pages 472–479, Honolulu, CA, USA, 2017. IEEE. doi: 10.1109/CLOUD.2017.67.

[147] S. Farokhi, P. Jamshidi, E.B. Lakew, I. Brandic, and E. Elmroth. A hybrid cloud controller for vertical memory elasticity: A control-theoretic approach. *Future Generation Computer Systems*, 65:57–72, 2016. doi: https://doi.org/10.1016/j.future.2016.05.028.

[148] *LXC container technology*, 2018 (Accessed April 15, 2014). URL http://www.ibm.com/developerworks/linux/library/l-lxc-containers.

[149] *OpenVZ Linux containers*, 2018 (Accessed April 15, 2014). URL http://openvz.org.

[150] V. Stankovski, J. Trnkoczy, S. Taherizadeh, and M. Cigale. Implementing time-critical functionalities with a distributed adaptive container architecture. In *Proc. of the 18th International Conference on Information Integration and Web-based Applications and Services*, pages 453–457, Singapore, Singapore, 2016. ACM. doi: 10.1145/3011141.3011202.

[151] S. Taherizadeh, A.C. Jones, I. Taylor, Z. Zhao, and V. Stankovski. Monitoring self-adaptive applications within edge computing frameworks: A state-of-the-art review. *Journal of Systems and Software*, 136:19–38, 2018. doi: https://doi.org/10.1016/j.jss.2017.10.033.

[152] S. Taherizadeh, I. Taylor, A. Jones, Z. Zhao, and V. Stankovski. A network edge monitoring approach for real-time data streaming applications. In *Proc. of the 13th International Conference on Economics of Grids, Clouds, Systems and Services (GECON 2016)*, pages 293–303, Athens, Greece, 2016. Springer. doi: https://doi.org/10.1007/978-3-319-61920-0 21.

[153] P. Hoenisch, I. Weber, S. Schulte, L. Zhu, and A. Fekete. Four-fold auto-scaling on a contemporary deployment platform using docker containers. In *Proc. of International Conference on Service-Oriented Computing*, pages 316–323. Springer, 2015.

[154] *Raspberry Pi 3 model B*, 2018 (Accessed April 15, 2014). URL https://www.raspberrypi.org/products/raspberry-pi-3-model-b/.

[155] *2017 State of the Cloud Survey*, 2018 (Accessed April 15, 2014). URL https://www.rightscale.com/blog/cloud-industry-insights/cloud-computing-trends-2017-state-\cloud-survey.